UNIVERSIDAD POLITÉCNICA DE MADRID

Escuela Técnica Superior de Ingenieros de Telecomunicación

Departamento de Ingeniería de Sistemas Telemáticos

ESTRATEGIAS DE PRUEBA DE LINEAS DE PRODUCTO DE SISTEMAS DE TIEMPO REAL ESPECIFICADOS CON DIAGRAMAS DE ESTADOS JERARQUICOS

Tesis Doctoral

Autor: Julio Mellado Torío
Ingeniero de Telecomunicación

Director de Tesis: Juan Carlos Dueñas López
Doctor Ingeniero de Telecomunicación

Año 2004

TESIS DOCTORAL

ESTRATEGIAS DE PRUEBA DE LINEAS DE PRODUCTO DE SISTEMAS DE TIEMPO REAL ESPECIFICADOS CON DIAGRAMAS DE ESTADOS JERARQUICOS

TRIBUNAL CALIFICADOR

Presidente:

-

Vocales:

-
-
-

Secretario:

-

Vocales Suplentes:

-
-

CALIFICACIÓN:

A mi madre, Josefina Torío Antich

Resumen

En la ingeniería del software el crecimiento de la capacidad tecnológica ha estado siempre unido con el incremento de la complejidad, y ello a su vez ha propiciado el nacimiento de nuevas técnicas para hacer frente a dicha complejidad. Las Líneas de Producto Software han aparecido en la ingeniería del software como una técnica cuyo objetivo es el de poder crear diferentes variantes software a partir de una infraestructura común, del mismo modo que se hace en otros sectores industriales.

La investigación en Líneas de Producto Software se ha centrado hasta la fecha en los aspectos de arquitectura software. Por medio de la "Ingeniería del Dominio" se definen los aspectos genéricos de todas las variantes y se especifican las entidades software genéricas denominadas activos básicos o "core assets", mientras que con la "Ingeniería de Aplicación" se establece el mecanismo de generación de las distintas variantes a partir de los elementos genéricos.

Un aspecto no menos importante, que hasta ahora no se ha investigado con tanta extensión, es el de las Pruebas de Línea de Producto Software. La cuestión fundamental es decidir hasta qué punto es posible probar las diferentes variantes de forma común. En el caso más optimista, probando una funcionalidad sobre la parte general se podría dar por probada sobre todas las variantes. Por contra, en el caso más pesimista, las pruebas de una Línea de Producto Software serían exactamente iguales que las pruebas de varios productos independientes que se hicieran de forma separada. Como término medio, aunque se pruebe la misma funcionalidad en todas las variantes, se podrían reutilizar por ejemplo la arquitectura de pruebas, los casos de prueba, el entorno de pruebas, etc.

Las Líneas de Producto Software pueden implantarse también para sistemas software de Tiempo Real. En este caso, los métodos de prueba propios de los sistemas de tiempo real se ven confrontados con la nueva exigencia de manejar la pruebas de las diferentes variantes.

Buscando dar una solución al problema de las pruebas de Líneas de Producto Software de tiempo real, la Tesis Doctoral propone un método de pruebas basado en los diagramas de estados jerárquicos o "statecharts" del lenguaje UML, que se utilizan como el modelo semiformal para definir los casos de prueba. Se propone una técnica para asegurar la correspondencia (trazabilidad) de los requisitos con los casos de prueba, estructurando los casos de prueba de forma semejante a los requisitos y estudiando cómo las variantes de los requisitos impactan sobre los diferentes elementos de prueba (modelos de prueba y arquitectura de pruebas).

También se define dentro del método un flujo de actividades, teniendo como objetivo la automatización de las pruebas como técnica apropiada para poder probar las diferentes variantes de la Línea de Producto Software de forma eficiente. La primera fase del proceso de prueba es el *Diseño de Pruebas*, donde se agrupan en *clases de prueba* los requisitos tanto genéricos como específicos de cada variante, y se modela cada clase de prueba mediante *diagramas de estados jerárquicos* y, si es preciso, *escenarios* en los cuales se incluyen de forma explícita sus *requisitos* asociados. En la fase de *Implementación de Pruebas*, estos diagramas se describen por medio de tablas de estados y eventos (soportados por una hoja de cálculo, por ejemplo), con la información adicional necesaria (los *datos de prueba*) para generar automáticamente el código con los casos de prueba ejecutables. En la fase final

de *Ejecución de Pruebas*, se realiza el caso de prueba sobre el sistema real y se registra el resultado obtenido.

Como aportación final, se propone un *metamodelo* donde se muestran todos los elementos del método de pruebas y las relaciones que existen entre ellos.

Para comprobar que la solución propuesta al problema de las pruebas de Líneas de Producto Software de Tiempo real es satisfactoria, se ha elegido una doble estrategia consistente en la aplicación del método sobre un caso industrial real y en la elaboración de un conjunto de herramientas software prototipo, con las cuales se ha demostrado la validez del método propuesto y se ha delimitado su alcance.

El caso industrial real escogido ha sido el de una Línea de Productos Software Tiempo Real de sistemas de control del tráfico ferroviario, en el que el autor desarrolla desde hace diez años su actividad profesional, lo cual da a esta Tesis Doctoral un marcado carácter industrial, tanto por la relevancia práctica del tema elegido como por reflejar una experiencia de aplicación industrial real.

Las herramientas software desarrolladas, tanto en el caso de aplicación industrial real, como en el ámbito de la Tesis Doctoral, sirven de soporte a la generación de casos de prueba a partir de los modelos de diagramas de estados, la ejecución automatizada de las pruebas, el análisis de los resultados o veredictos de las pruebas y la medición de cobertura de requisitos alcanzada en las pruebas.

Abstract

In the software engineering the growth of the technological capacity has been always united with the increase of the complexity, which has caused the birth of new techniques to face this complexity. The Software Product Lines have appeared in software engineering as a technique whose objective is the creation of different software variants from a common infrastructure, in the same way that is made in other industrial sectors.

The investigation in Software Product Lines has been centred until now in the aspects of software architecture. By means of "Domain Engineering" the generic aspects of all the variants are defined and the generic software parts, which are denominated "core assets", are specified; whereas with the "Application Engineering" the mechanism for the generation of the different variants from the generic elements is defined.

An aspect not less important, than until now has not been investigated in detail, is the Testing of Software Product Lines. The fundamental question is to what extent it is possible to test the different variants in a common way. In the most optimistic case, testing a functionality on the general part would mean consider that the given functionality has been tested for all the variants. On the other hand, in the most pessimistic case, the tests of a Line of Software Product would be exactly equal that the tests of several independent products. As intermediate approach, although the same functionality in all the variants has to be tested, some test artefacts could be reused i.e. the test architecture, the test cases, the test environment, etc.

The Software Product Lines can also be used for Real Time Systems. In this case, the test methods of the real time systems have to fulfil the new requirement of handling the test of the variants.

Looking for a solution to the problem of Testing of Real Time Software Product Lines, the Doctoral Thesis proposes a testing method based on the hierarchic state diagrams or "statecharts" of the UML language, that are used as a semiformal model of the implementation to define the test cases. A technique is proposed in order to assure the correspondence (traceability) of the requirements with the test cases, structuring them in a similar way as the requirements and analysing the impact of the variability within the requirements on the test artefacts (test models and test architecture).

A flow of activities within the method is also defined, having as goal the test automation as a proper way for testing the different variants of the Software Product Line efficiently. The first phase of the test process is the *Test Design*, where the requirements, (the generic ones and the specific ones of each variant) are grouped into *test classes* and each test class is modelled by means of a *statechart* and, if necessary, some *scenarios*, including the associated requirements explicitly. In the *Test Implementation* phase, the statecharts are introduced in form of a table of states and events into a spreadsheet with additional information (the *test data*) to generate the executable test cases automatically. In the last phase of *Test Exexution*, the test case is realised and the result of the test is registered.

Finally, a *metamodel* containing all the test artefacts and the relations that exist among them has been elaborated.

In order to verify if the proposed solution to the problem of the Real time Software Product Lines Testing is satisfactory, one double-way strategy has been chosen: the application of the method on a real industrial case and the elaboration of a set of prototype software tools, which demonstrates the validity of the proposed and delimits its scope.

The selected real industrial case has been a Real Time Software Product Line of control systems for railway traffic, which is a domain where the author has been carrying out since ten years his professional activity. The practical relevance of the subject chosen and the reflecting of an experience of real industrial application gives to this Doctoral Thesis a noticeable industrial character.

The developed software tools, as much in the case of real industrial application, like in the scope of the Doctoral Thesis, serve as support to the generation of test cases from the models of state diagrams, the automated execution of the tests, the analysis of the results or verdicts of the tests and the measurement of coverage of requirements reached in the tests.

Agradecimientos

Quisiera agradecer su colaboración y apoyo a todas las personas que han hecho posible la culminación de esta Tesis Doctoral:

En primer lugar, a Juan Carlos Dueñas, por su sabia labor de dirección de esta Tesis Doctoral, y por todos estos años de cooperación y trabajo.

A los profesores, investigadores, y personal del departamento de Ingeniería de Sistemas Telemáticos de la Universidad Politécnica de Madrid, con especial mención de Rodrigo Cerón y Tomás Robles.

A la compañía Alcatel, en especial a la persona de mi jefe José del Valle Alvarez, Director de I+D de la División de Automatización de Transporte, por su respaldo a este proyecto, concretado en mi participación como ponente en diversos foros técnicos presentando resultados de la compañía, que se ha reflejado también en la Tesis Doctoral.

A Begoña Sánchez, por su ayuda en el marco de su Proyecto de Fin de Carrera en la parte práctica de la Tesis Doctoral.

A mi familia y mis amigos, por su apoyo y consejos.

Contenido

Lista de Figuras

Lista de Tablas

1 Introducción y Objetivos

1.1 Motivación de la Tesis Doctoral

Tradicionalmente, el punto de vista predominante en la ingeniería del software es el desarrollo de sistemas individuales, donde se fabrica cada sistema software sin pensar en posibles sistemas similares futuros. La Reutilización del Software sería simplemente copiar para el nuevo sistema parte de los anteriores (requisitos, código, diseño, etc.), haciendo adaptaciones, si es el caso. Esta estrategia es por naturaleza asistemática, pues se reutilizan elementos que no han sido diseñados para volverse a utilizar en otros sistemas, lo cual acarrea ineficiencias en el desarrollo, código duplicado y un mayor esfuerzo de mantenimiento. Posteriormente, ha aparecido en el mundo de la ingeniería del software la idea de construir componentes software universales y reutilizables en múltiples contextos sin necesidad de grandes adaptaciones. Este es un objetivo que hoy por hoy no se ha alcanzado aún, como ha mostrado el fallo del Ariane 5 [Lio02].

Entre estos dos extremos, ha surgido un prometedor término medio, que son las Líneas de Productos Software, en las que se hace un diseño enfocado a la reutilización dentro de un dominio determinado con una funcionalidad restringida y para un número limitado de productos. Este concepto no ha sido fruto de las investigaciones teóricas, sino de la práctica. Algunas compañías europeas como Philips, Siemens, Nokia y Alcatel han organizado su desarrollo según este enfoque y las ideas en las que se basa se han formalizado en el concepto de desarrollo de software orientado a líneas de Producto, que está respaldado por organizaciones de desarrollo como el Software Engineering Institute de la Universidad Carnegie Mellon, los laboratorios de investigación Avaya [Gep02] (los antiguos laboratorios Bell), el European Software Institute (ESI), el Fraunhofer Institute for Experimental Software Engineering (IESE). El resultado ha sido reflejado en varias publicaciones [ClNo01], [Kuu00], [Du01], [Ler01], [CA03] y es un tema de discusión en foros dedicados al tema, como la International Software Product Line Conference (SPLC) o el Proyecto CAFE (from Concept to Aplication in System Family Engineering), donde se intercambian experiencias entre la industria y la comunidad investigadora. Los resultados son, en algunos casos, bastante alentadores, con algunas compañías indicando mejoras significativas en términos de esfuerzo, calidad y tiempo de entrega al mercado cuando se aplica la tecnología de Líneas de Producto Software.

La literatura de Líneas de Productos se ha centrado hasta ahora más en el desarrollo del software que en las pruebas de software. Está aún por aclarar si debe de utilizarse una estrategia de pruebas específica para Líneas de Producto Software o puede emplearse la misma estrategia de pruebas que para un sistema desarrollado de forma individual. Una estrategia de pruebas específica de Líneas de Producto Software tiene en cuenta la parte común de los diferentes elementos que la componen, de cara a optimizar el esfuerzo de pruebas. En este punto es importante el grado de modularidad del sistema, ya que si no existen interfaces internas claramente definidas y el grado de acoplamiento de los componentes software es muy alto, no es fácil poder afirmar que las pruebas pasadas sobre un producto valen para los demás.

Con el término estrategia de pruebas se quiere expresar la forma de definir las pruebas que se realizan sobre un sistema software partiendo de su especificación de requisitos. Una posibilidad es modelar dicho sistema software como un conjunto de máquinas de estados y recorrer de forma sistemática todos los estados y transiciones, ya sea de forma manual, o si se cuenta con medios para ello, de forma

automática. Ejemplos de pruebas de modelos con estados son [Bei95], [Hol99], [Hae99], [Bin00], [Chan98] y [MLS97], pero ninguno de ellos menciona aplicaciones de esta técnica sobre Líneas de Producto Software.

Otra de las carencias existentes en el terreno de las Líneas de Producto Software es la falta de herramientas software específicas para ayuda al Desarrollo y a las Pruebas. Es posible adaptar herramientas de propósito general para tareas parciales, pero hoy por hoy no existen herramientas comerciales que destaquen de forma clara. Ello es debido quizá a que el propio concepto de Línea de Productos Software está aún por asentarse de forma definitiva, y a que los entornos donde se aplica son muy variados.

1.2 Entorno de la Tesis Doctoral

Esta Tesis Doctoral pertenece al campo de la Ingeniería del Software, y dentro de éste a las Líneas de Producto Software. En los últimos cinco años ha habido un conjunto de proyectos europeos ESAPS y CAFE de investigación en este terreno, que han reconocido las Líneas de Producto Software como un terreno de valor estratégico, pero que todavía no está suficientemente dominado por los procesos de desarrollo software actuales, que están más centrados en los productos individuales.

El Departamento de Ingeniería Telemática (DIT) de la Universidad Politécnica de Madrid (UPM) ha participado activamente en estos proyectos, existiendo una línea de trabajo actualmente en este campo, que incluye aspectos como análisis de arquitecturas de Líneas de productos [DuOl98][Alo98][CaDu03], Métodos de Líneas de Producto [Cer00] y modelado de la variabilidad [CaDu01].

Paralelamente a la realización del doctorado en el programa del DIT de la UPM, el autor de la Tesis Doctoral ha trabajado en la compañía Alcatel, en la división de automatización de transporte (TAS), como ingeniero de software, realizando las siguientes tareas:

- 1993-1994 Encargado de la transferencia de tecnología en Alcatel TAS de una parte del enclavamiento electrónico ENCE L90 en colaboración con Alcatel Alemania.
- 1994-1995 Responsable de desarrollo de software para el sistema de control ferroviario ENCE L90 de la parte de control de elementos de hardware en distintos proyectos en España, Portugal, Polonia e Israel.
- 1995-1998 Participación en la especificación, diseño e implementación del enclavamiento electrónico INTERSIG 905.
- 1998-2000 Migración de plataforma hardware y del sistema operativo del sistema INTERSIG 905 colaborando con Alcatel Austria.
- 2000-2002 Responsable de las actividades de verificación de software en el departamento de enclavamientos electrónicos de Alcatel TAS.
- 2002-2004 Jefe de Departamento de Sistemas ATP (Automatic Train Protection) encargado de la transferencia de tecnología del sistema ETCS2000 en colaboración con Alcatel Alemania.

La experiencia en el entorno empresarial del autor ha hecho posible aplicar los conceptos de Líneas de Productos Software en un entorno industrial, y también valorar su importancia. En la industria existe el problema que las Líneas de Producto Software pretenden resolver, que consiste en adaptar un producto genérico para los diferentes proyectos, lo cual muchas veces no se consigue sino a costa de mucho tiempo y esfuerzo.

La elección del tema de la Tesis Doctoral ha ido paralela a las experiencias que he ido teniendo a lo largo de estos años en mi actividad profesional. Recién comenzados los cursos de doctorado, los modelos basados en diagramas de estados jerárquicos despertaron mi interés cuando comencé con

otros compañeros a aplicarlos en el primer diseño del módulo de Control de Elementos de Campo del enclavamiento electrónico INTERSIG 905, que culminó con gran éxito. Posteriormente, cuando las exigencias de los estándares de seguridad y los proyectos internacionales en que se participaban, hicieron necesario una mejora sustancial de los procesos de validación, contamos con la colaboración de la consultora de Bilbao SQS para mejorarlos. Paralelamente, por medio de mi tutor, Juan Carlos Dueñas, tomaba contacto con la comunidad investigadora de Líneas de Producto Software, de la que el DIT de la UPM es miembro destacado. Desde principios de 2002 estoy en el equipo de especificación de requisitos del proyecto ETCS 2000 en Berlín (Alemania), donde he vuelto a encontrarme con las Líneas de Producto Software, desde el punto de vista de la ingeniería de producto y del análisis del dominio de aplicación. De toda esa actividad, y de los distintos foros técnicos y académicos en los que he participado, han ido surgiendo algunas publicaciones, [DM04], [EMM02], [MeET02], [MeDu01], y [MeDu00]. Esta Tesis Doctoral intenta recoger todas esas experiencias y hacer una síntesis, que espero pueda resultar de interés.

1.3 Objetivos de la Tesis Doctoral

1.3.1 General

Proponer una metodología para la prueba de Líneas de Producto de sistemas software de tiempo real

En [ClNo01] se habla de las Líneas de Producto Software como fenómeno de reciente aparición en el panorama de la industria del software y de su importancia por la drástica reducción de costes que suponen cuando se aplican con éxito.

Las pruebas de Líneas de Producto Software son aún un terreno por explorar. En la literatura sobre Líneas de Producto no se encuentran apenas referencias a estrategias de prueba que abarquen de alguna forma el conjunto de todos los elementos de una Línea de Productos Software determinada. Y esto no es algo irrelevante, pues podría lograrse en las pruebas un ahorro de costes análogo al que se consigue en el desarrollo.

El desarrollo de estas estrategias de prueba tiene un grado de dificultad similar a la definición de la arquitectura de una familia de productos. Hay que estudiar cada uno de los productos que componen la familia y determinar qué tienen todos ellos en común, pero ese esfuerzo se ve recompensado porque el tiempo de desarrollo de cada uno de los miembros de las familias de productos se acorta. El objetivo de la estrategia de prueba para la línea de productos será, análogamente, probar de forma más fácil y efectiva que si se probara cada producto por separado.

Cuando se dice que el objetivo es proponer una metodología, se quiere decir que se pretende analizar las características de las pruebas de software en general en las diferentes fases del ciclo de vida para determinar qué es lo que hace falta específicamente para las pruebas de una familia de productos. El modelo de ciclo de vida de software de partida es el ya clásico modelo en V de la Figura 1.1. [Ren97]

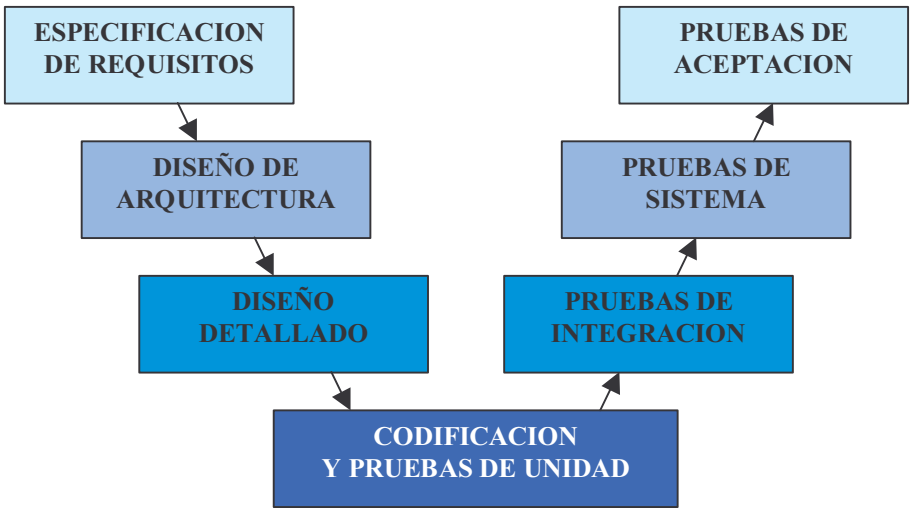


Figura 1.1 Modelo en V del proceso de desarrollo software

La metodología deberá ser aplicable ante todo en las pruebas de sistema, pues dicha fase es la principal de las pruebas y a la que se suelen dedicar más recursos, ya que verifican que se cumplen los requisitos del sistema. Las distintas variantes en los requisitos, que a su vez generan diversos productos, producen también variaciones en las pruebas, que se van a intentar abordar con la metodología objetivo de la Tesis Doctoral.

La metodología no va a abordar el resto de fases de prueba, con el fin de acotar el ámbito de estudio y poder obtener mejores resultados, no porque no se vea teóricamente posible el definir una metodología de pruebas para Líneas de Producto Software en dichas fases de prueba. Si en la línea de productos software existen componentes o módulos que tienen variaciones de un producto a otro, eso afecta a las pruebas de unidad. En las pruebas de integración, se prueban las interfaces del sistema y si la línea de productos admite variaciones en las interfaces del sistema, hay que contemplar esas variaciones en los casos de prueba de integración. En cuanto a las pruebas de aceptación del sistema, que tienen como objetivo [Ren97] "verificar que se cumplen los requisitos del usuario" y "se realizan en el entorno real del sistema", cualquier método que se proponga va a tener que considerar las que estas pruebas se tienen que definir conjuntamente con el cliente, y en la medida que existan diferentes clientes y entornos con requisitos comunes, es factible abordarlos de forma global con una estrategia de pruebas común a todos ellos.

Como referencias para la estrategia de prueba se toman los principios generales enunciados en [Mye79], [Perr95], y en las recomendaciones para pruebas de sistemas con estados de [Bei95] y de [Bin00]. También se han considerado las experiencias de [Due97] y [Ren97] específicas para pruebas de sistemas de tiempo real. Los sistemas de tiempo real que se van a considerar en el caso de estudio práctico se encuadran dentro de los denominados como *firm real time* en la literatura [Dou99]. Los requisitos de los sistemas *firm real time* son más relajados que los llamados *hard real time*, en los que los datos tardíos son datos erróneos. Para un sistema *firm real time* los requisitos de tiempos críticos tienen que cumplirse solamente en término medio respetando una desviación máxima, y, no supone por tanto un error, que en una ejecución particular el plazo crítico se sobrepase.

Se va a considerar en el método de pruebas de los sistemas de tiempo real objeto de la Tesis Doctoral únicamente los aspectos funcionales del sistema que se quiere probar, obviando todo lo relativo a la planificación de tareas y concurrencia, gestión de memoria, etc. Estos aspectos se van a probar de modo indirecto, cuando se verifique que el sistema de tiempo real responde a los estímulos externos dentro de lo prescrito por sus requisitos funcionales.

1.3.2 Específicos

Utilizar los diagramas de estados jerárquicos (statecharts) como el mecanismo básico de especificación de requisitos y generación de casos de prueba. (Objetivo Metodológico)

Demostrar la viabilidad de la metodología propuesta aplicándola en un caso práctico y comprobando que satisface unos objetivos de calidad estándar. (Objetivo de Validación)

Diseñar una herramienta para automatizar al máximo posible la ejecución de las pruebas y la medición de su grado de cobertura. (Objetivo Práctico)

La metodología de pruebas que se va a proponer utiliza como punto de partida de las pruebas una especificación de requisitos del sistema basada en diagramas de estados jerárquicos (statecharts). Los requisitos funcionales del sistema (incluyendo requisitos de tiempo real no crítico) se modelan con diagramas de estados jerárquicos, teniendo cada requisito uno o varios statecharts asociados y los requisitos funcionales del sistema que no pueden capturarse mediante statecharts de forma sencilla, se modelan con escenarios. Los requisitos no funcionales (rendimiento, calidad, seguridad, etc.) se dejan fuera del ámbito de la metodología. Los requisitos se van a clasificar en función de su grado de alcance dentro de la familia de productos. Pueden afectar a toda la línea de productos o sólo a unos miembros determinados. Es un objetivo de la Tesis Doctoral proponer un método de gestión de esta diversidad de los requisitos a la hora de las pruebas.

Los statecharts se eligen en vez de los diagramas de estados convencionales porque tiene mayor riqueza expresiva y son más fáciles de entender en cuanto crece la complejidad de la máquina de estados. Los aspectos referentes a métodos formales de los statecharts no entran dentro de los objetivos de la Tesis Doctoral. Se va a usar siempre como referencia la notación UML para los statecharts, sin tener en cuenta las diferentes variantes que se han propuesto.

A partir del modelo basado en statecharts de la especificación de requisitos, se elaborará una relación de casos de prueba. Posibles criterios para esta tarea son: recorrer las transiciones, recorrer todos los estados definidos, etc. Esos casos de prueba se traducen a un formato ejecutable por una herramienta, que los lleva a cabo y obtiene una medida de la cobertura de las pruebas (cuánto se ha probado del sistema total). El modelo de requisitos de línea de producto se va a corresponder con la especificación de casos de prueba del sistema, para que la metodología propuesta sea coherente.

El caso de aplicación seleccionado para demostrar la validez del método es una familia de productos de tiempo real de control de tráfico ferroviario con requisitos críticos de seguridad, concretamente de enclavamientos electrónicos, dominio en el que el autor desarrolla su actividad profesional desde el año 1993. Una caracterización muy completa de los requisitos de los sistemas de tiempo real "crítico" en general está recogida en [Dou99] y en [BIN97]. Desde el punto de vista de aspectos generales de implementación de estos sistemas es muy completo [Hat95]. Las referencias específicas de sistemas de seguridad que se van a manejar en el ejemplo son la norma CENELEC [CEN97] de desarrollo de software para sistemas de señalización ferroviaria, y los estándares de codificación para C y C++ [EBA94] y [EBA99].

La familia de productos que se va a usar como ejemplo tiene como requisito en las pruebas de sistema el cumplimiento de los criterios de calidad que define el estándar CENELEC, que es un estándar de alcance europeo para desarrollo de software con requisitos críticos de seguridad en el dominio de los ferrocarriles y cuyo cumplimiento se exige a los suministradores de sistemas por parte de las administraciones ferroviarias. Para este dominio, el estándar CENELEC define objetivos de calidad

(lo que se debe de garantizar con las pruebas) y da una serie de pautas, algunas obligatorias y otras sólo recomendadas, para las distintas fases del proceso de desarrollo software. Para el nivel de seguridad más crítico, los objetivos de calidad son la cobertura total de los requisitos (todos los requisitos definidos se cumplen) y del código (todas las líneas de código se han ejecutado durante las distintas fases de pruebas), que son objetivos de calidad aplicables a otros dominios análogos donde se usan sistemas de tiempo real con requisitos de seguridad.

El ejemplo escogido va a permitir evaluar la metodología mediante un ejemplo industrial real. Es adecuado para demostrar la validez del método porque el comportamiento de esta familia de productos se puede modelar mediante diagramas de estados. La literatura muestra que los diagramas de estados y los statecharts se usan en dominios muy variados y eso habla a favor del método propuesto, pues su campo de posible aplicación es bastante amplio. El ejemplo va a servir también para ayudar a fijar las ideas en el diseño de la herramienta objetivo de la Tesis Doctoral.

Se pretende desarrollar un mecanismo de generación de casos de prueba a partir de un statechart. Hay que comprobar primeramente que dicho modelo es correcto y que no existen los fallos en la definición del statechart que se refieren en [HaPo98]. El siguiente paso es traducir la representación gráfica del diagrama de estados a texto para tener, por cada caso de prueba un conjunto de acciones elementales a realizar sobre la aplicación bajo prueba para comprobar que su comportamiento es conforme a lo especificado en el statechart, en forma de fichero (*script*) de comandos o similar.

La herramienta es necesaria para ejecutar los casos de prueba de forma automática y almacenar el resultado de los mismos. Se reduce el coste de las pruebas, pues es posible repetirlas con menos esfuerzo que si se hiciera manualmente. Esta herramienta aportaría como novedad el gestionar por separado los casos de prueba comunes a toda la línea de productos y los casos de prueba específicos de cada producto.

La estructura de la herramienta de pruebas, similar a la de [Hol99], consiste en un “entorno de prueba” sobre el que se ejecuta la aplicación a validar. El entorno de prueba ejecuta los casos de prueba que se han generado a partir de los statecharts que expresan los requisitos del sistema: lleva al sistema al estado inicial, provoca el evento definido en el caso de prueba y compara la salida con el resultado esperado. Se registran las salidas del sistema y se comparan con los resultados esperados, midiendo el grado de cobertura de requisitos de las pruebas realizadas, para verificar que al ejecutar todos los casos de prueba de todos los statecharts hemos probado el cien por cien de los requisitos del sistema

1.4 Estructura de la memoria

El capítulo 2 ofrece una visión del estado de la ciencia en el campo de las Líneas de Producto Software y de las Pruebas de Sistemas Software. Dentro de las pruebas de Sistemas Software se habla de las Pruebas basadas en Diagramas de estados jerárquicos y en escenarios y se da una visión del estado actual de la Automatización de Pruebas.

En el capítulo 3 se expone el método de prueba de Líneas de Productos que se propone, basado en la trazabilidad de los requisitos con los casos de prueba que se obtienen a partir de modelos expresados mediante diagramas de estados jerárquicos, y que además está orientado a automatizar las pruebas.

El método del capítulo 3 se aplica en el capítulo 4 en un caso de estudio industrial real para demostrar su validez.

En el capítulo 5 se da una visión de las herramientas software de apoyo para implantar el método de pruebas.

El capítulo 6, presenta a manera de conclusiones, un resumen de las aportaciones de la Tesis Doctoral y un esbozo de las líneas de investigación que han quedado abiertas a partir de este trabajo.

2 Estado de la Ciencia

2.1 Líneas de Producto Software

2.1.1 Introducción

Las Líneas de Producto son una estrategia ampliamente utilizada en numerosas ramas de la industria que consiste en la producción de múltiples productos utilizando una infraestructura común. Este enfoque se está introduciendo también en la ingeniería del software, en la medida en que ésta va siendo cada vez más una actividad industrial que necesita mejorar su eficiencia y optimizar sus costes.

Una línea de productos software es un conjunto de soportes lógicos que poseen unas características comunes y que están orientados a las necesidades específicas de una determinada área de negocio [Vin99]. Los miembros de la línea de productos están construidos a partir de unos componentes software reutilizables (llamados activos básicos "*core assets*" en la literatura [ClNo01]) con el fin de que el desarrollo sea más breve y menos costoso, con mayor calidad y más predecible. El enfoque en línea de productos maneja de forma global el conjunto de los componentes de la familia de productos software.

Los "activos básicos" son, según [SEI01], todo aquello que constituye la base de la línea de productos. "Activos básicos" son la arquitectura, los componentes reutilizables, los modelos del dominio, los requisitos, la documentación y las especificaciones, los modelos de rendimiento, la planificación, los presupuestos, los planes de prueba, los casos de prueba, los planes de trabajo y las descripciones de proceso. El principal activo básico es la arquitectura. El conjunto de activos básicos se puede llamar también plataforma.

[SEI01] distingue entre línea de productos y familia de productos. La familia de productos, no sólo tiene características comunes, sino que está formada a partir de activos básicos. En este documento ambos términos tienen el mismo significado.

No todos los desarrollos de software pueden llegar a ser Líneas de Producto. La condición básica es que exista un conjunto de sistemas con algo en común. Luego, en función de las características del dominio y del nivel de riesgo e inversión que quiera asumir la organización, como apunta [Ban00], esos productos software llegarán a constituir una línea de productos o no.

Las Líneas de Producto, según [ClNo01] se apoyan sobre tres pilares: el desarrollo de activos básicos, el desarrollo de productos y la gestión. El desarrollo de activos básicos tiene como resultados la definición del alcance de la línea de productos, la arquitectura de la línea de productos y el resto de los activos básicos (todo aquello que sea común a varios miembros de la familia de productos), junto el plan de producción de los miembros de la familia de productos a partir de los activos básicos. La gestión es fundamental para conseguir que la política de reutilización intensiva de software que implica una línea de productos se lleve a cabo y sea asimilada por la organización.

Se va a comenzar exponiendo brevemente las ventajas que pueden hacer aconsejable la adopción de las Líneas de Producto Software. A continuación, se va a estudiar el precio que se ha de pagar por obtener esas ventajas, esto es, el impacto que las Líneas de Producto tienen sobre el proceso de desarrollo de software en general. En ese punto, se va a plantear el problema que tiene relación directa con el objetivo general de la Tesis Doctoral: ¿Definir familias de productos es más ventajoso para la fase de pruebas de software que el no hacerlo? ¿Hay alguna forma de probar las Líneas de Producto Software diferente a las pruebas de software de productos individuales? La respuesta a esa pregunta va a ser el método de prueba que se fijó como objetivo metodológico de la Tesis Doctoral. Se va exponer el método y su alcance, y se va a comparar con otros enfoques existentes. Una vez expuesto el método de pruebas, se plantea la posibilidad de hacer una parte de ellas de forma automática y se analiza si es viable.

Las fuentes bibliográficas que se manejan proceden principalmente de dos importantes grupos de trabajo sobre Líneas de Producto de software que existen actualmente, que son el del "Software Engineering Institute" (SEI) de la Carnegie Mellon University en Pittsburgh (EEUU), y los proyectos de investigación ESAPS (Engineering of Software Architectures, Processes and Platforms for Product-Families EUREKA-ITEA) y CAFÉ (From Concept to Application in System Family Engineering, EUREKA-ITEA), que aglutinan a departamentos de distintas universidades, entre ellas el DIT de la UPM, y centros de investigación de carácter público y privado dentro de la Unión Europea.

2.1.2 Concepto de línea de productos software

Una línea de productos software, también llamada familia de productos en la literatura, es un conjunto de sistemas software de una misma área de negocio y que tienen una cierta funcionalidad en común. La ingeniería de Líneas de Producto busca aprovechar esta parte común para desarrollar de forma eficiente y sistemática nuevos miembros de la familia de productos. Este enfoque, que no es ninguna novedad en otros ámbitos de la industria, está empezando a aplicarse en la ingeniería del software, por las ventajas de ahorro de costes que conlleva.

Mediante la ingeniería de Líneas de Producto, las organizaciones pueden reducir su esfuerzo de desarrollo, acortar los tiempos de llegada al mercado de nuevos productos, facilitar su mantenimiento y evolución, además de poder planificar de forma conjunta el desarrollo y el mantenimiento.

Ejemplos prácticos de ello son [Ame00], [Omm00], [Mac96] y [MSRD00]. [Ame00] describe cómo se ha especificado en Philips Medical Systems una familia de productos de aparatos de rayos X. Se ha realizado un modelo de los requisitos dividiéndolos en requisitos comerciales, de sistema y funcionales, añadiendo además un modelo de objetos en UML. Este modelo de objetos en UML recoge las distintas posibilidades existentes dentro de la familia de productos por medio de mecanismos de herencia, multiplicidad variable en las relaciones entre las clases y diferentes valores de los atributos. [Omm00] explica cómo se ha organizado el software empotrado de equipos de electrónica de consumo de Philips, estructurándolo en componentes que se pueden combinar de diferentes maneras para constituir diferentes productos. La arquitectura de referencia es siempre la misma: una capa de acceso al hardware, otra capa de procesamiento de señal de vídeo y audio y una tercera capa de servicios y aplicaciones. Dentro de cada capa hay distintos componentes, que se seleccionan para construir el producto deseado. [MSRD00] en cambio, cuenta cómo Alcatel ha desarrollado una plataforma para ejecutar sobre ella las nuevas versiones de aplicaciones ya existentes con requisitos críticos de seguridad y disponibilidad en el dominio del control de tráfico ferroviario. Una característica común de todos estos ejemplos es que el desarrollo en Líneas de Producto surge como respuesta frente a la complejidad y al alto coste. [Mac96] afirma que para que sea rentable, debe de haber al menos tres productos diferentes en la familia.

Un modelo de una línea de productos captura los requisitos de todos los productos dentro de un dominio determinado. Se modelan los requisitos comunes a todos los productos y los requisitos específicos de cada producto. Esta coincidencia en algunos de los requisitos y en el dominio es típica de las Líneas de Producto [MSRD00]. Están orientadas al mercado, en vez de estar orientadas a un cliente determinado [Dol98].

[Per98] afirma que tener una línea de productos implica tener una arquitectura genérica a partir de la cual se derivan de alguna forma todos sus componentes. Tiene que ser posible, afirma, tanto abstraer una arquitectura general a partir de los componentes de la familia, como derivar los diferentes productos individuales a partir de la arquitectura general.

[Nor00] muestra de forma práctica cómo realizar la arquitectura de una línea de productos de sistemas de control en tiempo real centrándose en tres aspectos: que la arquitectura sea extensible, esto es que se le puedan añadir y configurar nuevas capacidades fácilmente; que sea portable y que los cambios de plataforma software (sistema operativo, mecanismos de comunicación e interfaz de usuario) sean fácilmente realizables; y por último, que garantice que siempre se satisfagan los requisitos de tiempo real críticos.

2.1.3 Los diagramas de estados en las Líneas de Producto Software

Con las familias de productos, las empresas intentan sistematizar el desarrollo de software para hacerlo más efectivo. Una de las técnicas que pueden utilizarse en esta tarea, y que por otra parte, es algo de uso general en el desarrollo de software, es el empleo de los diagramas de transición de estados en las distintas fases del ciclo de vida, especialmente en la representación de los requisitos y en la formulación de los casos de prueba. Un ejemplo industrial de esta estrategia es [Jep00], que define el comportamiento común a todos los componentes de la línea de productos software mediante una máquina de estados, que luego se implementa como un componente aislado y reutilizable.

La arquitectura de la línea de productos puede, en el caso de que se deba de hacer “ingeniería inversa”, elaborarse a partir de sistemas ya existentes, lo que se denomina “recuperación” de la arquitectura: el trabajo de [KTW98] es un ejemplo de generación de componentes genéricos, en forma de máquinas de estados, que se extraen a partir de código fuente existente.

2.1.4 Ventajas de las Líneas de Producto Software

[Kna01] expone que hoy por hoy no existen datos cuantitativos de organizaciones y proyectos reales que apoyen la conveniencia de usar la tecnología de Líneas de Producto Software. Desde un punto de vista cualitativo [Kna01] afirma, apoyando cada una de sus hipótesis con un gráfico, que hay siete argumentos en favor de las Líneas de Producto:

- Reducen el esfuerzo de desarrollo dedicado a cada producto
- Reducen el tiempo de entrada en el mercado de cada producto
- Mantienen el tiempo de resolución de las peticiones del cliente constante, y no proporcional al número de productos.
- Permiten producir más funcionalidades en la misma cantidad de tiempo.
- Permiten producir más funcionalidades con la misma cantidad de dinero.
- Se reduce el tiempo de integración de Componentes Estándar (COTS, del inglés *“Commercial Off the Shelf”*) por producto.
- Se reduce el tiempo de certificación por producto, dependiendo de los requisitos de certificación necesarios y del organismo certificador.

[Ban00] propone un modelo de análisis para justificar la introducción de un desarrollo basado en Líneas de Producto en una organización. Las variables que maneja este modelo para cuantificar los beneficios de las Líneas de Producto son las inversiones necesarias, los gastos de mantenimiento de infraestructura derivados de la implantación de la línea de producto, y, por último, el ahorro de costes obtenido. Con estos tres parámetros se calcula el retorno de la inversión (ROI). Aparte de este cálculo, [Ban00] hace un análisis de riesgo de la implantación de Líneas de Producto considerando la adecuación para la reutilización de software en la estructura de la organización, el personal, los procesos y los productos. El resultado del análisis de riesgo (nivel de riesgo) se expresa en forma de porcentaje. El análisis se completa caracterizando la actitud frente al riesgo de la organización. Esta actitud se representa de forma gráfica en una curva como la de la Figura 2.1. Si el punto definido por el valor del ROI y del nivel de riesgo cae por encima de la curva de nivel de riesgo admitido, las Líneas de Producto son viables en la organización.

Figura 2.1 Riesgo frente a Retorno

[Sch01] describe un modelo económico de las Líneas de Producto. A diferencia de otros modelos económicos que se centran en el ahorro de costes por medio de la reutilización del software [Wil99], tiene una perspectiva más global. El modelo de [Sch01] tiene tres niveles de refinamiento. El primer nivel, que es el que [Sch01] describe con mayor detalle, se ocupa de los diferentes compromisos asociados al análisis económico de una línea de productos y en determinar el alcance óptimo de la infraestructura de reutilización del software. Los factores que [Sch01] tiene en cuenta son el esfuerzo de desarrollo, los recursos de personal disponibles (si los recursos son escasos, el esfuerzo de desarrollo para tener Líneas de Producto es mucho mayor en tiempo) y el grado de innovación del mercado (en mercados con muchos cambios, los beneficios de las Líneas de Producto apenas se notan). El segundo nivel considera los aspectos financieros asociados a las Líneas de Producto: el dinero gastado o ganado a fecha de hoy no tiene el mismo valor que el que se pueda gastar o ganar en el futuro. El tercer nivel se ocupa del análisis de riesgos y oportunidades de las Líneas de Producto, considerando por ejemplo la posibilidad de que aparezcan nuevos productos que no puedan ser incorporados a la línea de productos que se ha desarrollado, haciéndola obsoleta y disparando los costes.

Los autores mencionados hasta ahora coinciden en que no siempre va a ser conveniente implantar Líneas de Producto Software, y que siempre lleva asociado un cierto coste. Hay, sin embargo, puntos de vista complementarios, como los de [Ban01] y [Kru01] que afirman que la implantación de Líneas de Producto es posible con un coste mínimo mediante la reingeniería de los productos existentes. En ambos casos esta reingeniería se realiza con el soporte de tecnologías que ellos han ideado. [Ban01] se basa en el análisis del código fuente de los distintos productos. [Kru01] ha realizado una herramienta que utiliza los servicios de un sistema de gestión de configuración para confeccionar las distintas instancias de la familia de productos. Esta herramienta puede trabajar de tres formas, extractiva

(analiza las diferentes variantes de software ya existentes y separa lo común de lo específico), reactiva (reacciona frente a la inclusión de nuevos requisitos para los nuevos productos, redefiniendo las características de la parte común) y proactiva (partiendo desde cero, se define una parte común para los diferentes productos, y a partir de ella se derivan los componentes de la línea de productos).

2.2 El Desarrollo en las Líneas de Producto Software

Las Líneas de Producto Software representan un paso en la evolución del software de una actividad artesanal ([Br95] compara al ingeniero de software con el poeta) a un proceso industrial. Las Líneas de Producto se sustentan en una estrategia para la organización, la gestión técnica y los métodos de ingeniería de software. Por tanto, puede decirse que modifican el proceso de desarrollo software y le plantean unas exigencias determinadas. Ahora bien, ¿De qué naturaleza son esas exigencias? ¿Es tan diferente el proceso de desarrollo de una familia de productos del de un sistema individual? Evidentemente, dependerá del sistema, pero también es cierto que se han identificado algunas características propias del proceso de desarrollo software de una línea de productos.

Un ejemplo, desde el punto de vista práctico, es el trabajo de [Pos01], que recoge las experiencias del desarrollo con plataformas (los activos básicos de una línea de productos) dentro de la empresa Philips, desde el punto de vista del proceso de desarrollo software. El proceso de transición ha sido bastante costoso y no exento de riesgos, y ha implicado un cambio total del modelo de negocio. Aún así, los resultados han sido bastante positivos y se ha podido concurrir a más mercados que sin el uso de plataformas. [ClNo01] contiene multitud de experiencias en el ámbito industrial de Líneas de Producto Software.

En este apartado se va a hacer referencia a trabajos sobre el ciclo de vida de las familias de productos en general, profundizando posteriormente en un aspecto, la ingeniería de requisitos de las Líneas de Producto. En este último subapartado se va a introducir el concepto de trazabilidad, que va a ser importante en el método de pruebas que se va a proponer en este documento.

2.2.1 El ciclo de vida de las Líneas de Producto Software

[Cer00] realiza una comparativa de diferentes procesos de desarrollo de software de Líneas de Producto partiendo del proceso unificado definido por la casa Rational (RUP) [Kru03]. Las tres características más importantes de RUP son, según [Cer00]:

- El desarrollo está centrado en la arquitectura que se usa para "entender el sistema, organizar el desarrollo, fomentar la reutilización y evolucionar el sistema"
- Iteración o desarrollo en espiral: Las fases de desarrollo rígidas del modelo en cascada se reemplazan por una sucesión de pequeños ciclos en los que se va añadiendo funcionalidad paulatinamente.
- Basado en casos de uso: Cada iteración comienza definiendo que nuevos casos de uso (basados en requisitos) se van a implementar.

En RUP no se menciona a las Líneas de Producto Software en ningún momento. Eso quiere decir que los aspectos de desarrollo de software de Líneas de Producto deben de incorporarse de otros procesos que sí las tienen en cuenta. En [Cer00] se cita a PRAISE, SELECT, RSEB y a FORM.

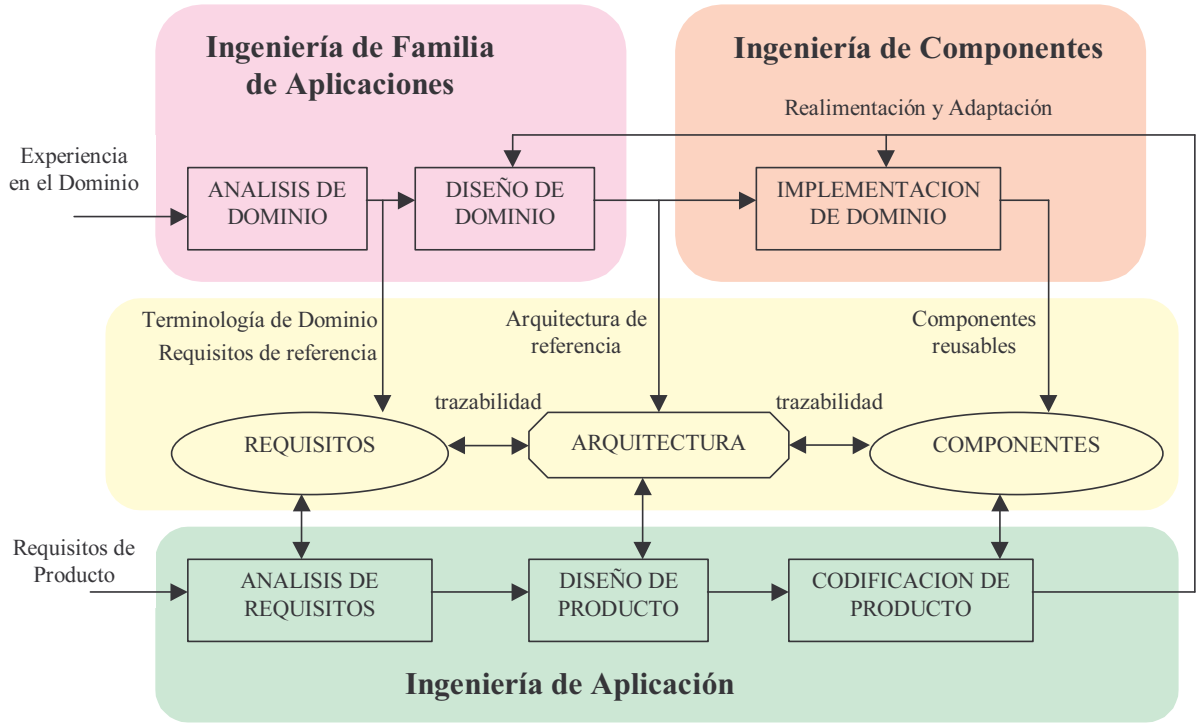


Figura 2.2 Proceso de desarrollo PRAISE

PRAISE (Product-line Realisation and Assessment in Industrial Settings) es un proceso definido en el ESI (European Software Institute) conjuntamente con Thomson y Bosch en el marco del proyecto europeo ESAPS [Lin00] que propone un esquema parecido al ciclo de vida en cascada, añadiéndole una fase de análisis de dominio cuyo resultado es la elaboración de una arquitectura de referencia. La implementación tiene como resultado la producción de componentes reutilizables.

El proceso de desarrollo SELECT hace hincapié en el uso de componentes, definiendo dos grandes flujos de trabajo, la producción de la aplicación final mediante la selección de componentes, y, la producción de componentes reutilizables.

RSEB divide las tareas de ingeniería de software en ingeniería de componentes, ingeniería de aplicación y en ingeniería de familia de aplicaciones. Como puede verse, están citadas por orden de menor a mayor grado de generalidad.

FORM (Feature-Oriented Reuse Method) establece dos procesos básicos que son la ingeniería de dominio y la ingeniería de aplicación. La ingeniería de dominio "son las actividades de análisis de los sistemas de un determinado ámbito cuyo resultado es la creación de un conjunto de componentes reutilizables y de una arquitectura de referencia". Tanto la arquitectura como los componentes son capaces de soportar los aspectos comunes del dominio como los específicos de una aplicación determinada.

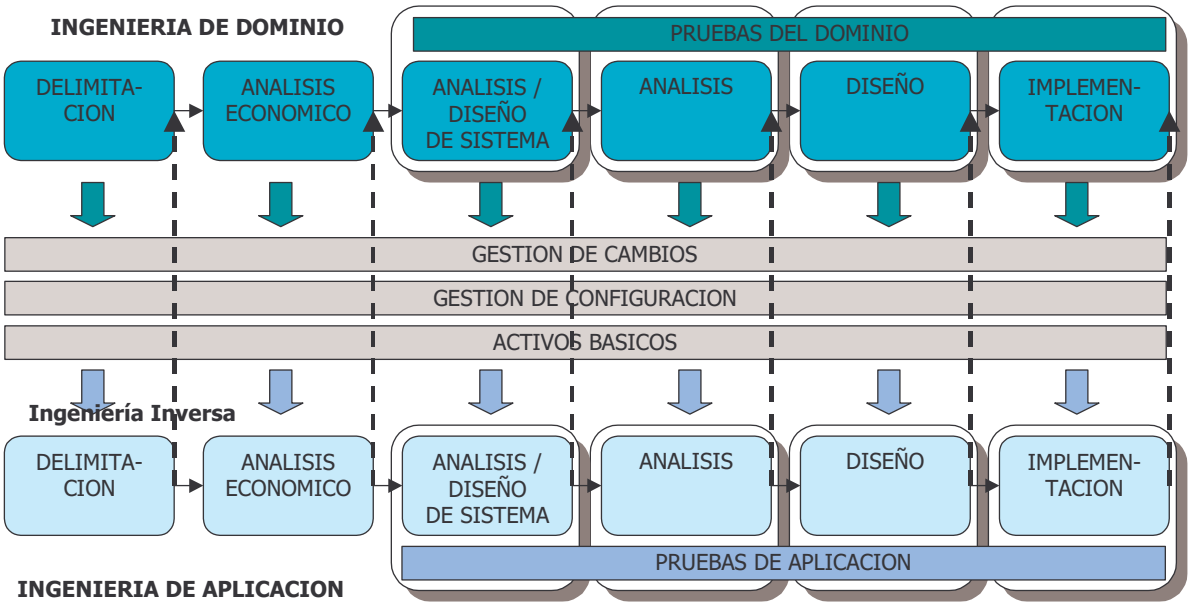


Figura 2.3 Actividades de Ingeniería en el Modelo de Referencia CAFE

El proyecto europeo CAFE (*"From Concepts to Application in System-Family Engineering*") es la continuación del proyecto ESAPS. Uno de sus resultados [CAF01] es el Entorno de Referencia de Familias de Sistemas. [CAF01] que es una combinación de cuatro modelos diferentes a su vez:

- Ciclo de vida de una organización que trabaja en el campo de las Tecnologías de la Información: Ofrece una visión global de alto nivel de la organización, identificando dos actividades principales, el uso de la familia de productos (desarrollo, despliegue y mantenimiento) y el preuso de la familia de productos (planificación, análisis económico, delimitación, etc.).

- Entorno de Procesos ISO/IEC 15.504, que clasifica todos los procesos de una entidad en cinco grupos: Cliente- Suministrador, Ingeniería, Organización, Gestión y Soporte.

- Modelo de referencia CAFE (CAFE CRM), que define seis grandes paquetes básicos de trabajo en una línea de productos.

- Modelo de activos CAFE (CAFE ARM), que clasifica los diferentes activos de la línea de productos según el modelo de referencia anterior.

La Figura 2.3 muestra el modelo de referencia CAFE, que incluye de forma expresa las actividades de prueba de la línea de productos, tanto de la ingeniería del dominio como de la ingeniería de aplicación. El modelo de activos que aparece en la tabla, asocia los distintos tipos de entidades software a cada una de las fases del modelo de referencia. Los artefactos de prueba pertenecen a las fases que son propiamente de desarrollo software (análisis, diseño e implementación).

<table>
<tr><th>DELIMI-TACION</th><th>ANALISIS ECONOMI-CO</th><th>ANALISIS Y DISEÑO DE SISTEMA</th><th>ANALISIS</th><th>DISEÑO</th><th>IMPLEMEN-TACION</th></tr>
<tr><td colspan="6">Pruebas</td></tr>
<tr><td colspan="6">Gestión de Cambios</td></tr>
<tr><td colspan="6">Gestión de Configuración</td></tr>
<tr><td colspan="6">Documentación de Gestión (Estimaciones, Planificaciones, etc.)</td></tr>
<tr><td colspan="6">Diccionario</td></tr>
<tr><td rowspan="4"></td><td rowspan="4">Plan de Negocio</td><td>Requisitos de Sistema</td><td rowspan="3">Otros Requisitos</td><td>Modelo de Datos</td><td>Clases</td></tr>
<tr><td>Plataforma Hw. y Sw.</td><td>Interfaces</td><td rowspan="2">Componentes</td></tr>
<tr><td>Arquitectura del Sistema</td><td>Arquitectura Software</td></tr>
<tr><td colspan="4">Artefactos de Prueba (Planes de prueba, Casos de prueba, etc.)</td></tr>
<tr><td colspan="6">Otros documentos (económicos, marketing, desarrollo, formación, etc.)</td></tr>
<tr><td colspan="4" rowspan="4"></td><td colspan="2">Librerías</td></tr>
<tr><td colspan="2">COTS</td></tr>
<tr><td colspan="2">Open Source (Fuentes de libre distribución)</td></tr>
<tr><td colspan="2">Servicios Web</td></tr>
</table>

Tabla 2.1 Modelo de Activos Básicos Software proyecto CAFE

El Fraunhofer IESE (Institute for Experimental Software Engineering) es una de las principales organizaciones que investigan en Líneas de Producto en Europa. Ha ideado un proceso de trabajo y unas herramientas de soporte para desarrollar Líneas de Producto, el método PuLSE (Product Line Software Engineering) que ha aplicado en distintas empresas pequeñas y medianas en Alemania [Kna00]. La Figura 2.4 muestra los tres principales elementos que constituyen este método:

- Las fases del proceso son las etapas del desarrollo de la línea de productos: inicio, desarrollo, uso y evolución.
- Los componentes técnicos son herramientas que se emplean en cada fase del proceso. Aparecen en la figura con el mismo color y borde que la fase del proceso en la que se emplean.
- Los componentes de soporte permiten evaluar la calidad de una aplicación del método en un cierto entorno.

En principio, el desarrollo en Líneas de Producto no está ligado a un método de análisis y diseño de determinado. [Ste99] analiza si es ventajoso el utilizar UML para describir la arquitectura de las Líneas de Producto. De entrada, UML tiene la ventaja de ser un lenguaje de modelado muy conocido y soportado por numerosas herramientas. UML ofrece mecanismos de representación de la variabilidad que si es necesario pueden completarse definiendo estereotipos por parte del usuario. Por ejemplo, definir un símbolo que represente un punto de variación en la arquitectura. Sin embargo, la notación gráfica de UML no permite documentar completamente una línea de productos, entre otros aspectos, lo referente a su mantenimiento ni a la manera de instanciar los diferentes productos a partir de la arquitectura básica. Por lo tanto, hay que acompañar los diagramas de UML con documentación de otro tipo (texto, etc.) cuando sea preciso.

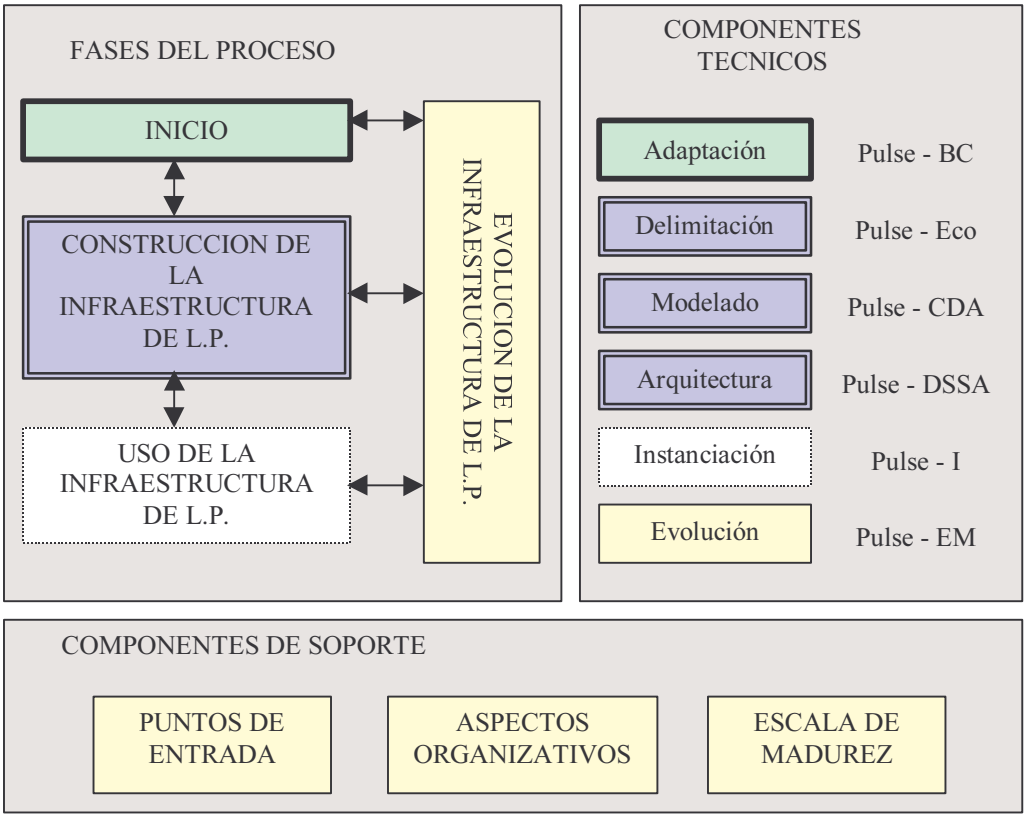


Figura 2.4 Método PulSE (Product Line Software Engineering)

[Tol01] dice sin embargo, que UML no es suficiente para desarrollar una línea de productos, pues no ofrece de forma explícita mecanismos que sirvan para modelar la variabilidad. Para [Tol01] utilizar UML u otro lenguaje de modelado similar para desarrollar familias de sistema es análogo a intentar implementar un sistema orientado a objetos con un lenguaje que no soporta tal orientación a objetos. [Tol01] propone trabajar con un mayor grado de abstracción, usando lenguajes específicos del dominio. Esto es, primero hay que hacer un análisis del dominio, identificando sus entidades y funcionalidades y luego desarrollar un conjunto de generadores de código y de herramientas software que permitan trabajar con el nivel de abstracción adecuado.

Al principio de cada uno de esos ciclos de vida lo que hay como entrada básica de ambos procesos de desarrollo son los requisitos. Los resultados del proceso de desarrollo de los productos son los propios productos. Ahora bien ¿Qué resultados tiene un desarrollo de familia de productos? Es claro que aparte de los diferentes productos, se elabora toda la infraestructura de reutilización de software que va a permitir obtenerlos con menos esfuerzo. [Vin99] propone un modelo de dominio de Líneas de Producto que incluye los siguientes elementos:

- Modelo de requisitos: Todos los demás elementos del modelo de dominio tienen que tener requisitos asociados.
- Modelo de contexto: define los límites del dominio
- Modelo de capacidades: es la descripción de la parte variable y la parte común de la línea de productos. Las capacidades opcionales de los elementos de la línea de productos están conectadas con los puntos de variación que existen en la arquitectura. Es un modelo elaborado según el método FODA (Feature Oriented Domain Analysis).

- Modelo operacional: especifica el comportamiento de forma más precisa que los casos de uso. Puede contener statecharts, diagramas de secuencia, etc. [Vin99] lo considera opcional, al igual que el Modelo de Casos de Uso.
- Modelo de arquitectura: describe el diseño de alto nivel de la línea de productos. [Vin99] propone que en este modelo se haga referencia a los patrones de diseño que se usen en la arquitectura

Sintetizando todo lo dicho hasta ahora, pueden hacerse las siguientes afirmaciones:

- La fase de análisis del dominio es muy importante en el desarrollo en Líneas de Producto, puesto que en ella se definen los aspectos comunes a todos los miembros de la familia y los particulares de cada uno.
- Hay que definir los activos básicos de la línea de producto (ingeniería del dominio) y la forma de instanciar cada producto particular a partir de dichos activos básicos (ingeniería de aplicación).

2.2.2 Especificación de requisitos

La ingeniería de requisitos de las Líneas de Producto es, obviamente, más complicada que la de un producto desarrollado de forma individual. Hay que tener en cuenta, como dicen [MaBr99] y [ClNo01], los requisitos de todos los componentes de la familia. [MaBr99] estudia la aplicación de los casos de uso de UML a la ingeniería de requisitos de una línea de productos y concluye que, aún utilizando las relaciones de inclusión, extensión y generalización, no son suficientes para representar completamente la variabilidad dentro de una línea de productos. Además, los casos de uso tampoco valen para modelar requisitos no funcionales. [ClNo01] propone usarlos contemplando "puntos de variación" para que puedan modelar la variabilidad en la línea de productos.

[ClNo01] explica que, en el análisis de requisitos, hay que encontrar similitudes e identificar variaciones. Esto es la base para posteriormente establecer los requisitos comunes a toda la línea de productos y los específicos de un producto. La verificación de requisitos se ve también afectada si se está hablando de una línea de productos, pues hay que comprobar que los requisitos están correctamente clasificados. Por último, la gestión de los requisitos ha de tener también estos aspectos en cuenta, a la hora de gestionar los cambios en los requisitos.

El método de análisis FODA [Ka90] desarrollado por el SEI (Software Engineering Institute) de la Universidad Carnegie Mellon, es uno de los principales métodos de partida en el terreno del desarrollo de Líneas de Producto. Examinando los diferentes sistemas software relacionados, el análisis de dominio permite obtener una descripción genérica de los requisitos y una forma de implementarlos. El resultado de aplicar FODA es una familia de productos más que un producto individual, produciendo un modelo del dominio con la flexibilidad suficiente para reflejar las diferentes posibilidades que existan, y una arquitectura estándar para desarrollar componentes software.

Las fuentes del análisis del dominio son, en FODA, la documentación publicada sobre el dominio (libros, etc.), los estándares, las aplicaciones existentes y los expertos. La Figura 2.5 recoge todas las fases y productos del análisis del dominio.

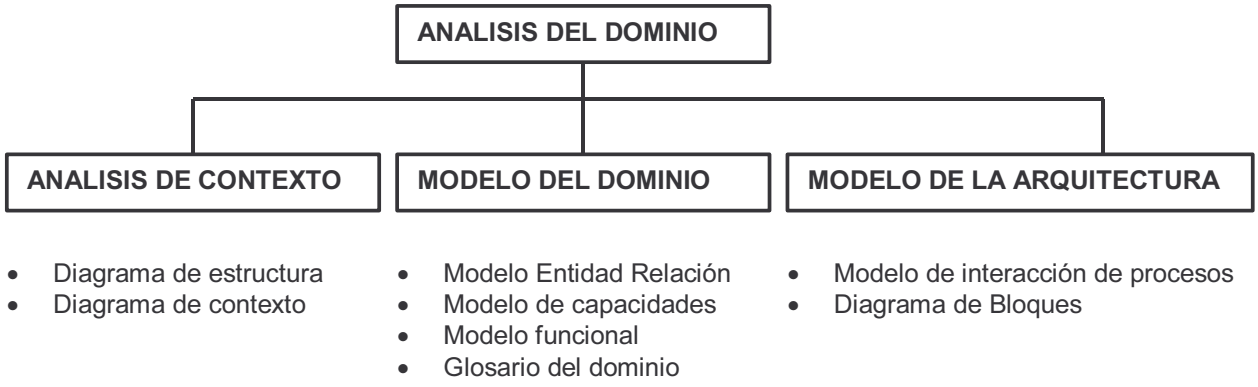


Figura 2.5 Productos y fases del Análisis del Dominio en FODA

Una de las aportaciones de FODA que tiene más relación con el problema objeto de esta Tesis Doctoral es el modelo de capacidades. FODA organiza las capacidades (lo que ve el usuario final) de la familia de productos constituyendo jerarquías en forma de árbol. Las capacidades o características comunes se colocan en los nodos raíz del árbol. Cada variante sobre estas características comunes implica una rama en el árbol. La ventaja de esta técnica es que se recogen en un único modelo todas las posibles variaciones que hay en la familia de productos y que se identifican claramente. Las capacidades pueden ser obligatorias, opcionales o alternativas (Ver el ejemplo sencillo de la Figura 2.6).

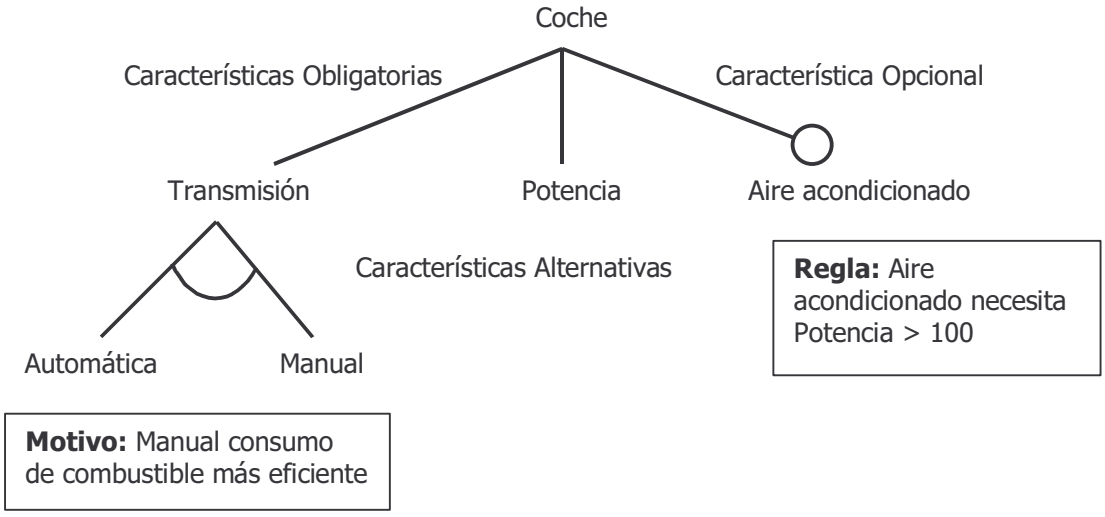


Figura 2.6 Ejemplo de modelo de características en FODA

[Kuu00] ha seguido profundizando en los resultados de FODA y propone un modelo de organización de requisitos para Líneas de Producto en forma jerárquica que contenga todos los requisitos de todos los miembros de la familia de productos. En la raíz del árbol estarían los requisitos básicos y comunes a todos los productos, incluyendo los requisitos no estrictamente funcionales que afectan a la globalidad de los sistemas (calidad, rendimiento, etc.).

Para confeccionar la jerarquía de requisitos, [Kuu00] distingue entre objetivos de diseño y decisiones de diseño. Los objetivos de diseño son cualidades que el sistema que se va a implementar debe de

satisfacer, con independencia de que sean muy generales o más concretas. Las decisiones de diseño reflejan las soluciones que se han dado para implementar el dominio, en otras palabras, la arquitectura de la línea de productos. El propio [Kuu00] llegado a este punto dice que esta práctica de mezclar requisitos y diseño de arquitectura no suena muy habitual, y que es bastante criticable. Pero afirma que en el caso de las Líneas de Producto, estas actividades no pueden separarse fácilmente, pues hay que estar seguros de que la arquitectura pueda soportar los requisitos de todos los productos, pues, en el caso contrario, se tiene un grave problema. Aun así, admite que al principio de la fase de requisitos, esto no es siempre lo más adecuado, y propone como solución de compromiso, documentar todas las decisiones de diseño que se hagan durante la fase de requisitos. En esto coincide con el modelo de desarrollo "twin peaks" [Nus01], que busca tender un puente entre la definición de requisitos y el diseño de la arquitectura. Los requisitos se definen en paralelo a la arquitectura de forma incremental. Esta estrategia tiene el inconveniente de que el proceso de desarrollo tiene una gestión más complicada, al no existir unos hitos claramente diferenciados

[Kuu00] estructura los requisitos, como ya se ha dicho, de forma jerárquica, al igual que FODA, situando los requisitos más generales en la parte de arriba del árbol (Ver el ejemplo en Figura 2.7, basado en [Kuu00]). Esto tiene la ventaja de especificar de forma concisa los objetivos generales y es un buen soporte para la fase de pruebas. Se da por probado el requisito raíz cuando se han probado todos los requisitos derivados. También permite solucionar conflictos de requisitos y eliminar las posibles inconsistencias. [Kuu00] ha desarrollado una herramienta que soporta este método, consistente en una base de datos de requisitos a la que se accede mediante consultas SQL para obtener la información que se desee en cada caso en varios formatos estándar a elegir (RTF, HTML, PDF y XML).

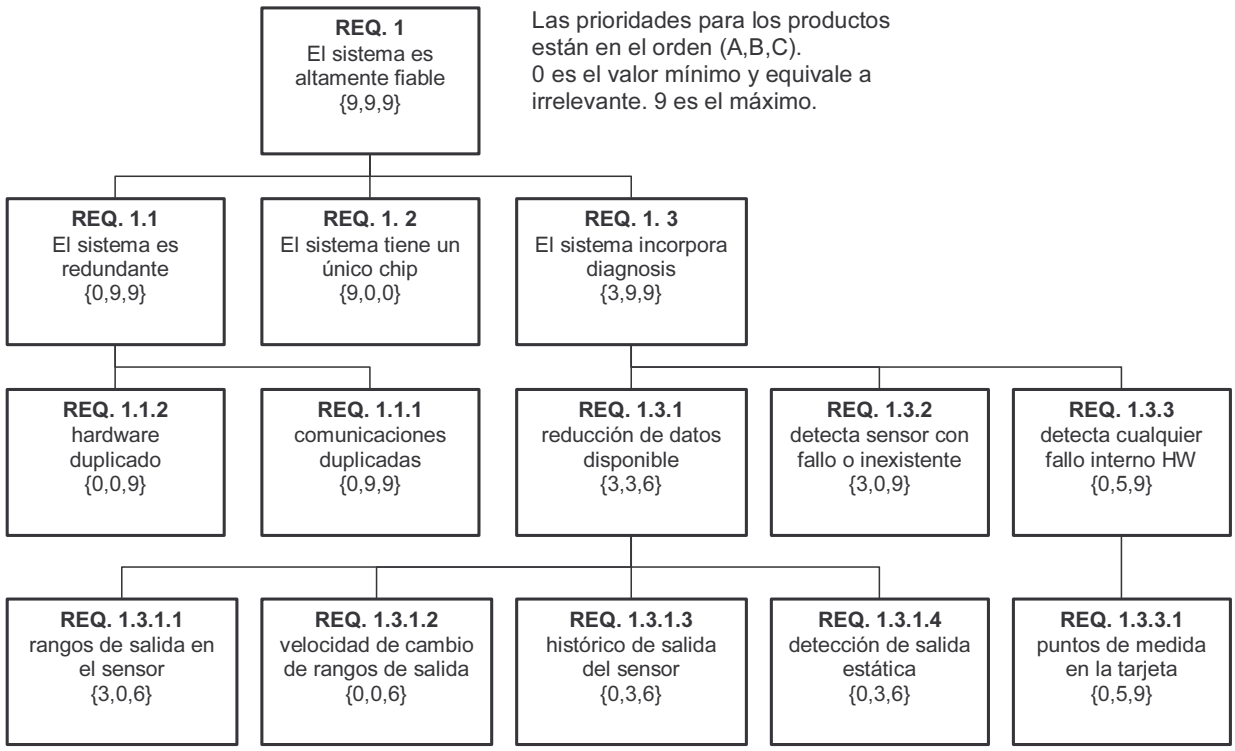


Figura 2.7 Ejemplo de jerarquía de requisitos

Un punto discutible de este método es la complejidad que trae consigo la gestión de prioridades con rango de 0 a 9. El caso más habitual será que el requisito sea obligatorio para un producto, o no lo sea. En ese caso bastaría tener nada más que dos valores, y podría obviarse todo ese tratamiento.

2.2.3 Trazabilidad de requisitos en Líneas de Producto Software

La consistencia entre los diferentes modelos y resultados de desarrollo es clave. Esta propiedad se denomina trazabilidad. [Bay01] define la trazabilidad como "posibilidad de seguir la evolución de un concepto a través del desarrollo de software": primero viene el requisito, luego la arquitectura, a continuación el componente, y al final, el código. Según [Bay01], la trazabilidad es clave para la coherencia de las diferentes etapas del desarrollo y en el mantenimiento de las Líneas de Producto. [Lut01] también mantiene la misma opinión, debido a que si hay trazabilidad entre requisitos y componentes, se facilita mucho obtener los diferentes productos. El nuevo producto tendrá nuevos requisitos, que tendrán otros componentes asociados. Esto dependerá de cómo estén diseñados los componentes.

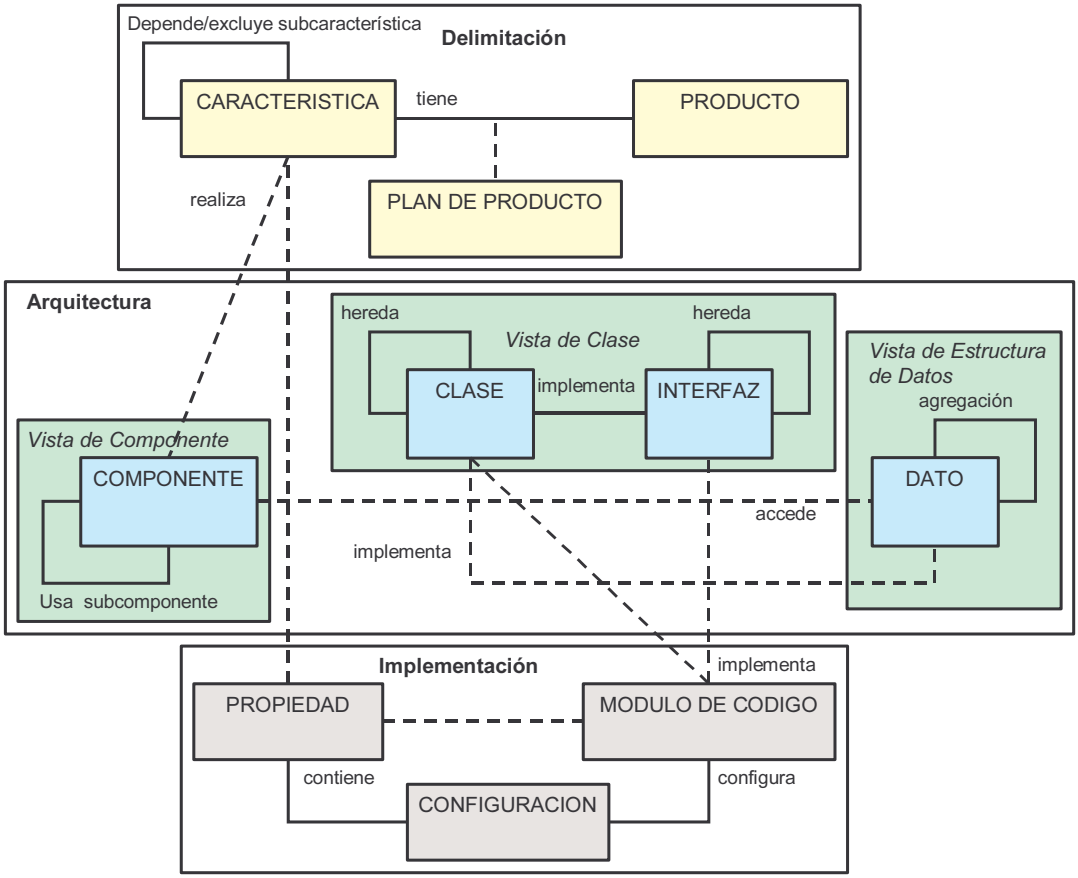


Figura 2.8 Metamodelo de trazabilidad

El mecanismo de trazabilidad de características del dominio en componentes de arquitectura que propone [Bay01] tiene los siguientes requisitos:

- Está basado en la semántica de los modelos que forman la infraestructura de la línea de productos. La traza no es un mero enlace sin significado.
- Admite diferentes variedades de traza.
- Es capaz de manejar la variabilidad de la línea de productos.
- El número de trazas es el mínimo posible.
- Es automatizable.

Para ello, utiliza en un primer paso un metamodelo o modelo genérico de la línea de productos donde se identifican las posibles trazas, fijándose especialmente en los aspectos comunes a los diferentes modelos (dominio, arquitectura, código, etc.). En la Figura 2.8 se ve el metamodelo, que recoge las tres fases del proceso de desarrollo de Líneas de Producto (delimitación, arquitectura, implementación) donde se ha centrado [Bay01].

En [Bay01] se refiere la aplicación de esta estrategia en el desarrollo de una línea de productos software de sistemas financieros. En cuanto a la automatización, explica que esta trabajando con la herramienta Rational Rose para generar los modelos y que utiliza una base de datos para capturar las relaciones de trazabilidad en los diferentes modelos. La estrategia de [Bay01] es analizar con la ayuda de herramientas los diferentes modelos que se pueden generar con Rational Rose e identificar los elementos que tienen el mismo nombre en varios modelos como trazas potenciales. [Bay01] no dice si para poder hacer eso se necesita seguir unas reglas de nombrado de los elementos del modelo, cosa que parece probable. El ámbito de automatización propuesto por [Bay01] no cubre la relación entre los requisitos y los casos de prueba del sistema (trazabilidad horizontal).

2.3 Las Pruebas de Software

2.3.1 Introducción

Las pruebas son un aspecto importante en el desarrollo de software. Precisan de una estrategia efectiva, con el máximo grado de automatización posible, con el fin de optimizar el coste que suponen.

El desarrollo en línea de productos supone un esfuerzo adicional en el diseño de la arquitectura, pues hay que conseguir una parte general adaptable y reutilizable en los diferentes sistemas. Ahora bien, esto no garantiza a priori que probando un producto individual estemos probando el resto de la familia de productos. Hay que determinar qué pruebas son generales para todos los miembros de la familia, y esto es un reto técnico importante.

El contexto en que usemos los distintos componentes de la arquitectura general de la línea de productos puede afectar su funcionalidad, especialmente si el grado de encapsulación no es muy alto y la interfaz del componente es muy compleja. Las distintas configuraciones admitidas en la línea de productos pueden variar completamente las condiciones de funcionamiento, y modificar por tanto las características de las pruebas.

[Bal98] dice que gran parte de la investigación y el trabajo práctico en el campo de las Líneas de Producto está englobado dentro de las siguientes tres áreas: arquitecturas de referencia, componentes reutilizables y generadores de componentes. Este autor apunta una cuarta línea de trabajo en infraestructuras para probar y depurar diferentes miembros de una familia de productos.

Estas infraestructuras de prueba serían una respuesta frente a la complejidad creciente del desarrollo de software y la reducción de los plazos de puesta en el mercado. La calidad de los productos de software, especialmente si tiene requisitos críticos de seguridad, no debe verse afectada. En esta línea, aunque no orientado específicamente a Líneas de Producto, [Sch00] describe un entorno integrado de gestión de las pruebas software. El entorno gestiona los datos del proyecto durante el desarrollo de software. Esto incluye, entre otras informaciones, requisitos, documentos de diseño, y casos de prueba, así como las relaciones existentes entre todos ellos.

En [Bay00] se refiere cómo la validación de arquitecturas de Líneas de Producto es más compleja que las arquitecturas convencionales y se muestra el proceso integrado de creación y evaluación de

arquitecturas de Líneas de Producto PuLSE-DSSA. La primera fase en este proceso es la creación de escenarios para capturar los requisitos más importantes de la familia de productos: casos de uso críticos, objetivos de calidad (requisitos no funcionales) y las restricciones principales. A continuación, se determinan los escenarios que tienen validez desde un punto de vista genérico dentro de la línea de productos y se clasifican según este criterio. Por cada escenario se genera un conjunto de casos de prueba, que se recogen en un plan de evaluación de arquitectura. Cuando se ha terminado de definir la arquitectura, se le pasan los casos de prueba para evaluarla.

2.3.1.1 Las pruebas de software en general

Se ha expuesto en qué manera las Líneas de Producto tienen un proceso de desarrollo particular. Esta discusión sirve para encuadrar la pregunta que se plantea a continuación, y que está directamente relacionada con los objetivos de la Tesis Doctoral. La cuestión no es otra que, dado que las Líneas de Producto, en determinadas ocasiones, reducen el esfuerzo de desarrollo ¿aportan también alguna ventaja en las pruebas?

Para responder a la pregunta, hay que aclarar que se entiende por pruebas del software, relacionándolas con las otras etapas del proceso de desarrollo. La Figura 2.47, tomada de [Ren97], muestra un modelo de ciclo de vida del software.

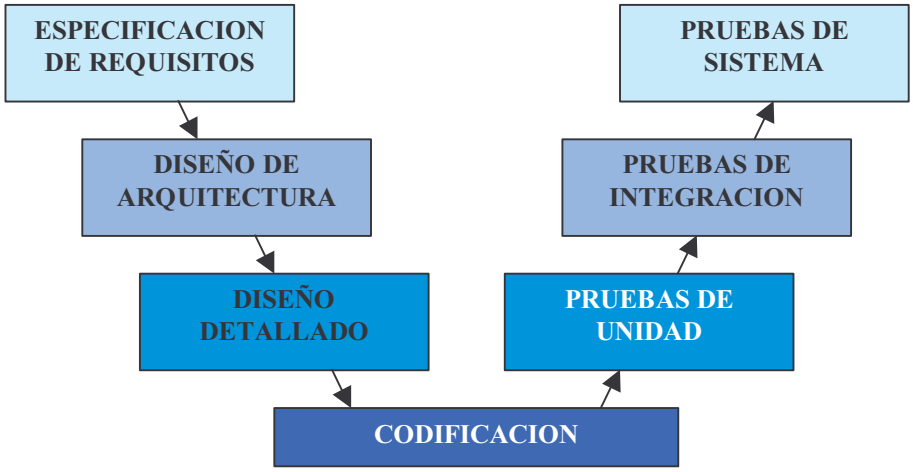


Figura 2.9 Modelo de ciclo de vida del software

El modelo muestra diferentes tipos de pruebas. Por un lado están las pruebas unitarias, que consisten en la prueba de diferentes partes o módulos de software por separado, y simulando de alguna forma los componentes que interactúan con el que se está probando. Las pruebas de integración consisten en, una vez que se han probado los distintos sistemas por separado, se prueban todos juntos, especialmente las interfaces. Tras esas pruebas de integración, se realizan las pruebas de sistema completo. No aparecen en la figura, pero tras las pruebas de sistema van las pruebas de aceptación.

[Ren97] define las pruebas de sistema diciendo que "una vez construida y probada la estructura del soporte lógico, estas pruebas establecen si el sistema completo funciona de acuerdo a las especificaciones. También se realizan pruebas para verificar requisitos no funcionales del sistema, como por ejemplo la robustez, la seguridad, la resistencia, la sensibilidad y el rendimiento".

Las pruebas de aceptación son realizadas en el entorno real del sistema, es decir, con los periféricos y usuarios. Su objetivo es verificar que se satisfacen los requisitos del usuario. Es habitual que se realicen conjuntamente con el cliente y que tengan carácter contractual, es decir, que la conformidad

del cliente con los resultados de las pruebas implique la finalización del proyecto y la entrega del software. Obviamente, es fundamental que el número y la severidad de los defectos que se descubran en esta fase de pruebas sean lo más reducidos posible.

Sin embargo, el modelo en V tiene una limitación que debe subrayarse. El modelo muestra el desarrollo de software como una secuencia de fases claramente diferenciadas. Esto es útil de cara a la planificación de los proyectos de software, pero no muestra el proceso incremental del desarrollo de software, a diferencia de otros modelos como el "modelo en espiral" de Boehm [Bo96]. Se planifica una funcionalidad reducida, se implanta y se prueba. La funcionalidad planificada para el sistema se alcanza de forma gradual en varios ciclos de desarrollo y prueba.

Este enfoque puede implicar un gran esfuerzo de pruebas de regresión al final del desarrollo, pues hay que decidir entre probar sólo los cambios realizados en el software respecto al último incremento probado, con el riesgo de dejar fallos sin descubrir; o bien, probar todo el sistema completo, que es costoso en tiempo y esfuerzo. En el campo de la técnica de pruebas de regresión, [Ki00] ha realizado un estudio empírico muy interesante sobre un grupo de programas ejemplo que ha modificado para introducirles errores, aplicando diferentes algoritmos de selección de casos de prueba (hacer todos, hacer un número aleatorio, hacer sólo los relacionados por los cambios, etc.). Sus resultados muestran que es más difícil detectar errores con un conjunto de casos de pruebas prefijado cuanto más cambios se hayan introducido en el software. En esa situación, seleccionar los casos de prueba aleatoriamente, es en términos relativos (defectos detectados por caso de prueba ejecutado) equivalente a ejecutar todos los casos de prueba. También concluye que la frecuencia con la que se hagan las pruebas de regresión influye en su efectividad.

Lo normal es utilizar una combinación de las dos formas de entender el ciclo de vida, planificando varias entregas del software con cada vez más funcionalidad y en las que la fase de pruebas tiene un tiempo asignado antes de cada entrega. En cualquier caso, la forma de abordar las pruebas del software está muy condicionada por el dominio de la aplicación que se está probando.

Como ejemplo de ciclo de vida diferente al modelo en V, [Red01] ofrece una visión complementaria, proveniente de la prueba de aplicaciones de Internet, donde los tiempos de desarrollo son muy ajustados. La Figura 2.10 muestra las etapas de las pruebas durante el desarrollo y la producción (cuando la aplicación está ya accesible en la Web). Es equivalente a utilizar una estrategia de prototipado rápido, pero procurando identificar cuanto antes los elementos más críticos del sistema en cuanto al rendimiento (algo que en las aplicaciones de Internet es muy importante). Hay datos reales sobre el rendimiento del sistema cuando se termina el desarrollo (Paso 4 de la Figura 2.10).

Una característica de las aplicaciones para Internet es, según [Red01] que cuanto más éxito tienen, más peticiones por parte de los usuarios hay para introducir nuevas funciones. [Red01] propone que se siga utilizando también en esta fase la técnica de modelar la nueva funcionalidad, construyendo un prototipo sencillo ya sea con una herramienta de simulación o con una hoja de cálculo, probar el prototipo para ver si satisface los requisitos de rendimiento y, si la prueba resulta positiva, comenzar el desarrollo. Si no, hay que rehacer el modelo hasta llegar a cumplir las prestaciones de rendimiento deseadas. [Red01] concluye, que con todo, esta estrategia de prueba es necesaria pero insuficiente en el mundo Internet, porque siempre puede haber problemas en la red (congestión, etc.) ajenos a la aplicación que hagan que funcione incorrectamente. Las pruebas deben de asegurar, que en caso de fallo, el sistema es lo suficientemente robusto como para comportarse de una manera predecible.

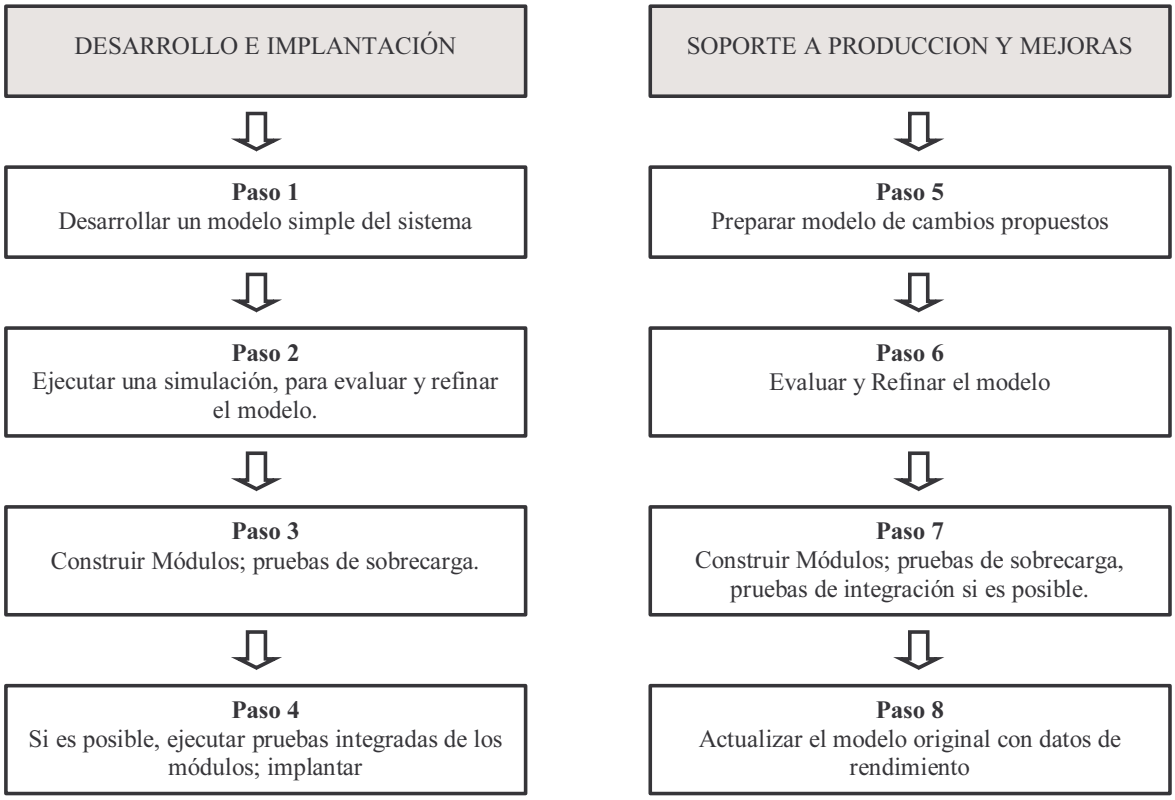


Figura 2.10 Pruebas de aplicaciones de e-business: desarrollo y soporte a producción

Otro ejemplo de ciclo de vida de software diferente al del modelo en V es la denominada "programación extrema" (XP, del inglés *"Extreme Programming"*) [Bec99]. En lugar de planificar, analizar y diseñar a medio o largo plazo, los programadores XP hacen todas estas actividades simultáneamente a lo largo de todo el desarrollo. La Figura 2.11 muestra una comparación entre los modelos de ciclo de vida en cascada, iterativo y XP. El método XP recomienda que:

- Los programadores escriban el código entre dos, para poderse corregir mutuamente.
- Inmediatamente después de desarrollar una parte de software, hay que escribir una prueba unitaria para comprobarla.
- Siempre hay que desarrollar la solución más simple.

Otro ejemplo de ciclo de vida alternativo al modelo en V son los desarrollos software "opensource": los desarrolladores son voluntarios, el código fuente es de libre disposición y la organización está descentralizada al máximo. [Ray98] recomienda a los coordinadores de este tipo de proyectos que consideren a las personas que prueban las versiones beta del software como uno de los recursos más valiosos del proyecto. En este tipo de desarrollos, se hace un uso intensivo de herramientas de gestión de defectos, habitualmente también "shareware", como GNATS y Bugzilla. La comunicación es primariamente por medio de correo electrónico [Cu98]. Pero sin embargo, no abundan los ejemplos de proyectos de este tipo donde se aplique una metodología de pruebas de forma sistemática. Se libera la versión, los desarrolladores que quieren probarla lo hacen e informan de los fallos. Otros desarrolladores, a su vez, se encargan de corregirlos. Ejemplos de proyectos software de este tipo que han tenido éxito son el sistema operativo Linux, el lenguaje Perl y las herramientas GNU. También hay productos opensource destinados a las pruebas como JUnit, una herramienta de pruebas unitarias, Deja GNU, un entorno de prueba automática, y OpenSTA, una arquitectura de prueba para sistemas distribuidos basados en CORBA.

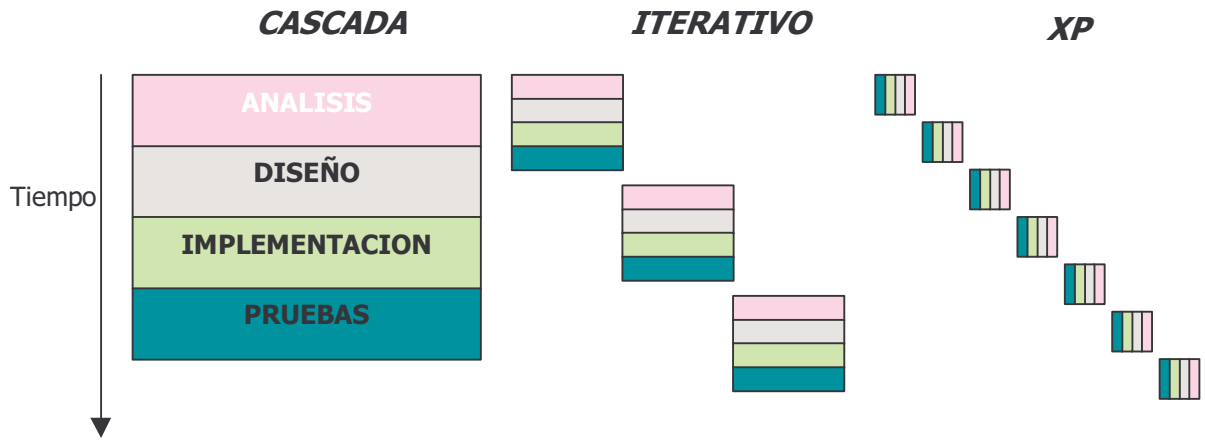


Figura 2.11 Ciclo de vida Software XP, comparado con el ciclo de vida en cascada y el iterativo

Estos ejemplos que se han citado anteriormente muestran como el modelo de ciclo de vida en cascada con las pruebas al final de todo ya no es válido. La fase de pruebas tiene que ser considerada en las fases anteriores del desarrollo. Una especificación de requisitos ambigua es difícil de probar, como también lo es una arquitectura no modular, o un código mal documentado. Las actividades de verificación de los resultados de cada una de las fases del ciclo de vida previas a las pruebas, son también importantes para detectar los defectos lo antes posible.

Esta consideración es el paso previo para dejar de ver las pruebas como una fase aislada dentro del ciclo de vida, y contemplarlas como un proceso paralelo al proceso de desarrollo. ¿En qué sentido? [GrSy01] afirma que las pruebas son un tipo de actividad que se aplica en varios momentos del desarrollo y no sólo al final del desarrollo y únicamente sobre el código. Para [GrSy01] las pruebas son un proceso separado pero íntimamente unido con el desarrollo, pero con objetivos muy diferentes y baremos de eficiencia muy dispares. Un número de defectos bajo en el desarrollo es considerado como algo positivo, mientras que en las pruebas es al contrario (cuantos más defectos se encuentren, mejor).

Las pruebas y el desarrollo forman un bucle realimentado (ver Figura 2.12), donde las salidas de uno son las entradas del otro. El desarrollo necesita la información de las pruebas para identificar las causas de los errores y corregirlos. Las pruebas también están condicionadas por los métodos y herramientas empleados en el desarrollo. La forma y calidad de los requisitos afectan tanto a pruebas como a desarrollo.

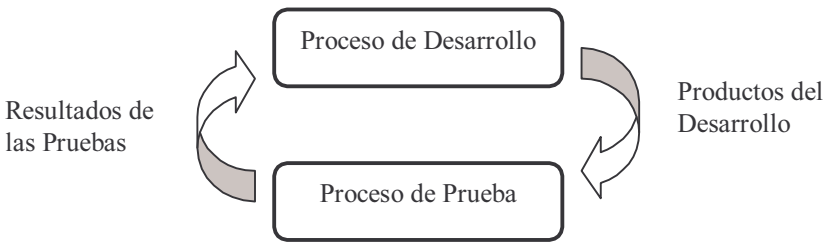


Figura 2.12 Bucle realimentado entre Desarrollo y Pruebas

Habría, según [Het84], tres fases en las pruebas de un proyecto software:

- *Análisis*: El producto que se va a probar es examinado para identificar las características que deben de recibir una atención especial a la hora de probarse y para determinar los casos de prueba que deben de realizarse.

- *Construcción*: En esta fase se desarrollan los instrumentos necesarios para las pruebas. Los casos de prueba identificados en la fase de análisis se implementan por medio de lenguajes de programación, de scripts o herramientas de prueba. Es necesario crear los conjuntos de datos que van a usarse en las pruebas como entradas del sistema.

- *Ejecución y Evaluación:* Esta es la parte más visible y a menudo la única que es reconocida de todo el esfuerzo de prueba. Sin embargo es normalmente la más breve. Los casos de prueba identificados durante el análisis y que a continuación han sido construidos, se ejecutan y el resultado se examina para ver si la prueba ha pasado o ha resultado fallida. Gran parte de estas actividades puede automatizarse, lo cual es especialmente provechoso en un entorno iterativo donde las mismas pruebas hayan de repetirse sobre diferentes entregas de un mismo producto.

Por lo tanto, si en el desarrollo de software no puede obviarse o hacerse a la ligera sin grave riesgo el trabajo de análisis y diseño, en las pruebas de software ocurre exactamente lo mismo. Y ocurre lo mismo porque son dos procesos muy dependientes el uno del otro. Y es que las pruebas tienen su origen en los requisitos, al igual que el desarrollo. Tan importante es que el desarrollo no pierda los requisitos de vista, como que las pruebas los tengan en cuenta. Esto equivale a que el proceso de pruebas sea capaz de garantizar la trazabilidad entre requisitos y casos de prueba.

Hay dos posibilidades para planificar pruebas de caja negra al terminar el desarrollo: realizar esas pruebas de caja negra de acuerdo a un protocolo definido previamente, o dejar al criterio de un experto en el sistema la elección y el diseño de las pruebas que hay que realizar. La primera opción asegura que se prueba la funcionalidad básica del sistema, pero no permite conocer que porcentaje de los requisitos se ha probado. Es costosa cuando el sistema ha llegado a su madurez, ya que cualquier cambio en el software, por pequeño que sea, demanda un gran esfuerzo de pruebas de regresión. La otra estrategia, usando la capacidad de un experto haciendo pruebas de caja negra de forma aleatoria, puede encontrar fallos que un proceso sistemático detectaría muy difícilmente, pero tampoco es capaz de dar una medida de cobertura de las pruebas (no se sabe a priori que porcentaje de los requisitos se han probado, ni que porcentaje del código se ha ejecutado durante las pruebas). No hay una solución única. La forma de realizar las pruebas del software depende de muchos condicionantes como los plazos, los recursos disponibles y el tipo de sistema que se está probando. Pero está claro que si el esfuerzo de pruebas se dirige en la dirección equivocada, puede resultar muy ineficiente, quedando insuficientemente probados requisitos importantes del sistema frente a otros requisitos probados demasiado exhaustivamente.

A continuación se recogen algunas aportaciones de la literatura que dan una visión de las pruebas como proceso complementario al desarrollo y con un idéntico punto de partida, que son los requisitos. Esto va a servir posteriormente como referencia para definir el proceso de prueba aplicable a Líneas de Producto que la Tesis Doctoral tiene como objetivo.

2.3.1.2 Procesos de prueba software

Usando el proceso de desarrollo RUP (Rational Unified Process) como referencia, [Arm01] habla de cómo puede usarse UML para entender cómo debe de funcionar el sistema que se está probando y cómo UML puede servir para planear y diseñar pruebas. La siguiente tabla contiene una lista de los diferentes modelos de UML y su posible aplicación en actividades de prueba.

Modelo	Descripción
Modelo de Casos de Uso del Negocio	Contiene actores de negocio y casos de uso de negocio y sus especificaciones. El comportamiento puede expresarse con diagramas de casos de uso y diagramas de actividad. Se usa para identificar el contexto externo en el que el sistema va a utilizarse
Modelo de Objetos de Negocio	Contiene unidades organizativas, colaboradores y entidades de negocio. La estructura de la organización puede expresarse con diagramas de clases. Distintos diagramas (de actividad, de colaboración, etc.) muestran como las diferentes partes de la organización interactúan entre sí.
Modelo de Casos de Uso	Contiene actores y casos de uso. El comportamiento externo del sistema puede expresarse con diagramas de casos de uso y diagramas de actividad. Sirve para identificar los roles relevantes en cuanto a la interacción con el sistema. Es importante conocer qué casos de uso van a implementarse y en qué momento para planificar las actividades de prueba.
Modelo de Diseño	Contiene clases y subsistemas. El comportamiento interno se expresa mediante diagramas de secuencia y de colaboración. La estructura interna del sistema puede modelarse mediante diagramas de clases. Es crucial identificar cuáles son las interacciones críticas que deben de probarse. Se usa también para identificar cualquier subsistema de prueba y/o clase que deba de ser modelada y construida para la integración y prueba del sistema en las primeras iteraciones cuan do aún hay partes del sistema que no se han desarrollado.
Modelo de Datos	Contiene tablas y sus relaciones. La estructura de la base de datos puede expresarse usando diagramas de clase.
Modelo de Procesos	Contiene cómo se ejecutan las clases del sistema en procesos y/o hebras. Utiliza diagramas de clase. Es crítico identificar las partes del sistema con especiales requisitos de disponibilidad y fiabilidad para someterlas a pruebas de rendimiento
Modelo de Implementación	Contiene componentes y sus relaciones. Sirve para entender la estructura física del software mediante los diagramas de Módulos.
Modelo de Despliegue	Contiene procesadores y dispositivos. Representa con los diagramas de despliegue cómo se reparten los procesos entre las máquinas disponibles y cómo se comunican a través de una red. Sirve para identificar posibles cuellos de botella en el sistema.

Tabla 2.2 Lista de los modelos de UML y su posible aplicación en actividades de prueba

Para el diseño de las pruebas, [Arm01] propone utilizar los diagramas de actividad. Un único diagrama de actividad puede mostrar el flujo principal y los posibles flujos alternativos de un caso de uso. En cada flujo se pueden identificar casos de prueba. [Arm01] no da una regla precisa para realizar esta selección, sino que dice que se haga intentando obtener la máxima cobertura de requisitos posible con el menor número de casos de prueba. Los casos de prueba para un caso de uso, se incluyen en un paquete (es un elemento de modelado de UML equivalente a un directorio en un sistema de ficheros). Para modelar la trazabilidad, [Arm01] define el caso de prueba como un estereotipo de UML, como puede verse en la Figura 2.13. El caso de prueba es representado unido con una flecha a su caso de uso asociado. La flecha representa la relación entre el caso de uso y el caso de prueba.

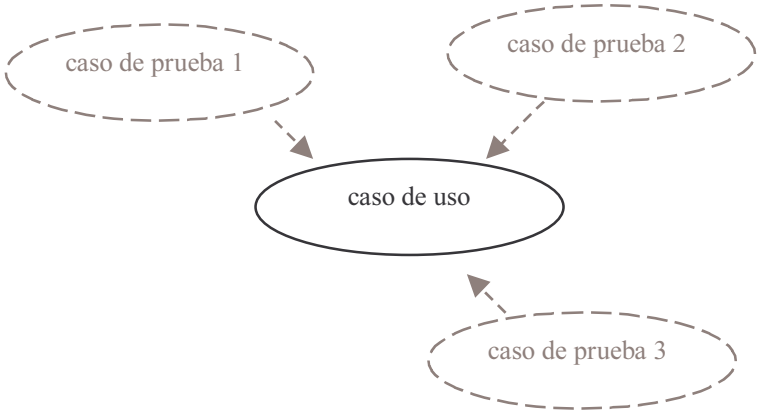


Figura 2.13 Diagrama de Trazabilidad de Casos de prueba a Caso de uso

Una vez que se han definido los casos de prueba, [Arm01] descompone cada caso de prueba en varias operaciones simples que denomina "procedimientos de prueba". El caso de prueba es una secuencia de llamadas a los diferentes "procedimientos de prueba". Esa secuencia puede estar controlada por una herramienta, si las pruebas son automáticas, o por una persona ([Arm01] lo denomina "guionista" del caso de prueba). Los procedimientos de prueba pueden reutilizarse en distintos casos de prueba, lo cual proporciona una gran flexibilidad para definir múltiples casos de prueba. La tabla que va a continuación muestra un ejemplo de un sistema de control de inscripciones para congresos. La Figura 2.14 muestra un diagrama de secuencia de uno de los casos de prueba definidos en la tabla.

Caso de Prueba	**Procedimiento de Prueba**
Registrarse en un congreso	Seleccionar Registrarse Seleccionar Congreso disponible Introducir información de participante válida Introducir información de pago válida Mandar Confirmación
Registrarse en un congreso completo y quedar en lista de espera	Seleccionar Registrarse Seleccionar congreso completo Introducir información de participante correcta Introducir información de pago correcta Mandar petición de lista de espera correcta Mandar confirmación
Registrarse en un congreso y no mandar la confirmación	Seleccionar Registrarse Seleccionar Congreso disponible Introducir información de participante correcta Introducir información de pago correcta No Mandar Confirmación

Tabla 2.3 Ejemplo de caso de prueba

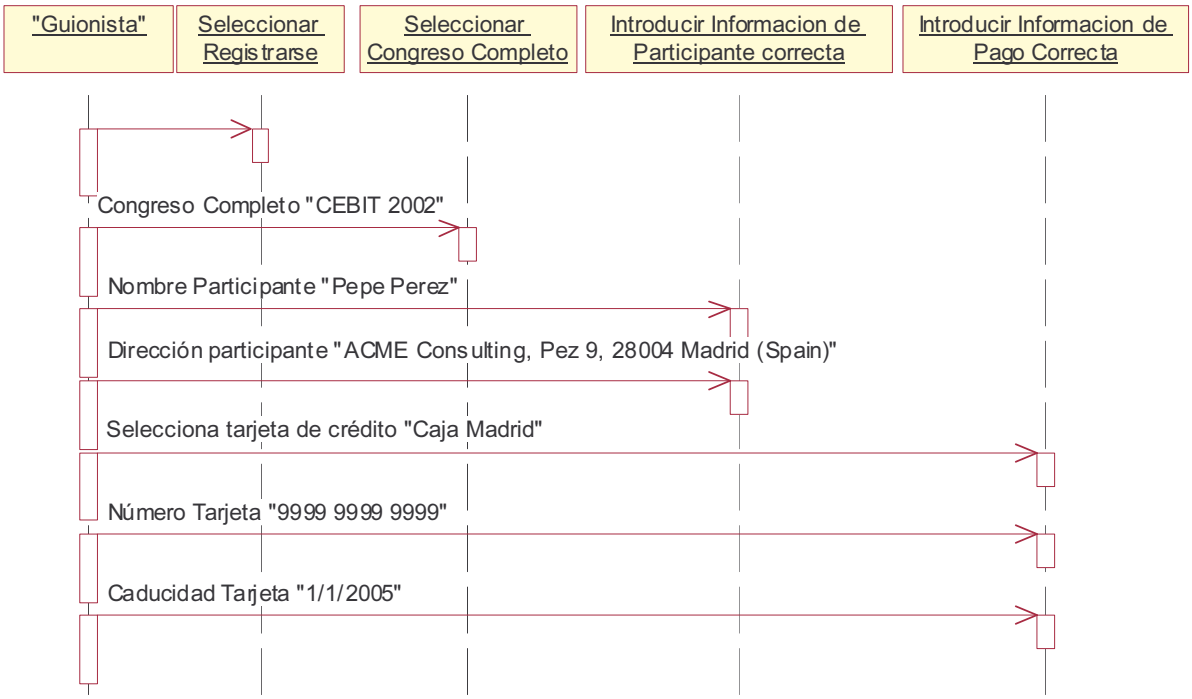


Figura 2.14 Diagrama de Secuencia Ejemplo de Caso de Prueba

Los procedimientos de prueba se modelan mediante un diagrama de clases, cuyos métodos sean las funciones que pertenecen al procedimiento. Como conclusión, la metodología de pruebas de [Arm01] no hace referencia a las Líneas de Producto, pero tiene aportaciones de gran interés que pueden ser útiles también en dicho campo: destacan la trazabilidad de los casos de prueba y el método de prueba de los escenarios. Y, si se usa UML en el desarrollo, su empleo permite a los ingenieros de prueba utilizar un lenguaje común al de los ingenieros de desarrollo.

Un ejemplo práctico de definición de un proceso de prueba en el ciclo de vida completo del software y en el que las pruebas se han automatizado está recogido en [Sik01]. Se describe un sistema software configurable de control de pasos a nivel que ha sido desarrollado por la empresa Siemens y el proceso de prueba que se ha elaborado para este sistema de tiempo real. El punto de partida del desarrollo y de las pruebas es idéntico: la especificación de requisitos, que se traduce a modelos de autómatas que, puestos en forma de tabla, sirven de base para las pruebas automatizadas. Se relata también la experiencia con varios pasos a nivel piloto. Una vez puestos en servicio, estos pasos a nivel se conectan a un equipo de diagnosis que monitoriza su comportamiento (el estado de las señales de control del paso a nivel) y lo transmite a una base de datos centralizada para analizar su funcionamiento.

2.4 Modelos de prueba

Se ha hablado de la importancia de los requisitos como punto de partida para la formulación de casos de prueba. Normalmente, estos requisitos estarán formulados en un lenguaje no formal. A partir de esta información, es habitual que los analistas y las personas encargadas de las pruebas confeccionen modelos de carácter formal o semiformal que les sirven de ayuda para entender mejor el sistema software con el que están trabajando.

El modelo de pruebas debe representar correctamente los requisitos y reflejar adecuadamente la implementación a probar. Al desarrollar un modelo de pruebas obliga a recapitular acerca del análisis y el diseño y, a menudo, recurrir a técnicas de ingeniería inversa. Esta estrategia se denomina *model-based testing* en la literatura [ElF00].

Esta estrategia de pruebas basada en modelos es mucho más poderosa si los modelos están integrados en los requisitos del sistema y si está disponible una representación semiformal de los requisitos, ya que a partir de estos modelos abstractos se deriva el material de pruebas (condiciones de prueba, casos de prueba, etc.). La automatización de parte de este proceso puede añadirle un gran valor, pues hace que añadir o modificar el material de pruebas sea mucho menos costoso.

Este apartado va a centrarse en dos tipos de modelos que se van a emplear posteriormente:

- los diagramas de estados y,
- los escenarios o diagramas de secuencia.

Por eso se va explicar como abordar la prueba de cada tipo de modelo dedicándole un subcapítulo específico.

2.4.1 La máquina de estados

La ***máquina de estados*** representa un sistema cuya respuesta viene determinada no sólo por la entrada actual, sino también por las anteriores. Estas entradas pasadas se quedan reflejadas en el *estado*. En contraste, las salidas de un sistema combinacional dependen sólo del estado en el que se encuentre. Por ejemplo, un candado de los comúnmente llamados de combinación, para abrirlo es necesario marcar unos valores específicos en un orden concreto, y es la secuencia de las entradas lo que determina la respuesta del candado.

Estas máquinas son la aplicación en ingeniería de lo que matemáticamente se conoce como *autómata finito*. Una maquina de estados es una *abstracción* de un sistema real, compuesta por eventos, estados y acciones. Una máquina de estados se construye sobre cuatro componentes:

> *Estado*- Resume la información relativa al *pasado* del sistema necesaria para determinar el comportamiento del mismo frente a futuras entradas.
> *Transición*- Paso permitido de un estado a otro. Viene ocasionada por un evento y puede dar lugar a una salida. Debe especificar los estados entre los que se produce.
> *Evento*- Una entrada, un intervalo de tiempo que provoca una reacción en el sistema.
> *Acción*- La respuesta del sistema a un evento.

Conceptualmente el funcionamiento de una máquina de estados es muy simple:

- Se parte de un *estado* llamado *inicial*.
- Si no se producen alteraciones la máquina permanece en este estado indefinidamente.
- Se recibe un *evento.*
- Si el evento no afecta al estado es ignorado, permaneciendo la máquina en el mismo.
- Si el evento se acepta, se produce una transición que puede ir acompañada de una acción de salida, y la máquina pasa al *estado destino*, convirtiéndose éste en actual.
- El proceso se repite hasta alcanzar el *estado final,* que puede existir o no.

Hay que tener en cuenta que la máquina sólo puede tener un estado activo en cada instante, y es requisito imprescindible para cambiar de estado que se produzca una (y sólo una por vez) transición. Los eventos se reconocen de uno en uno, siempre y cuando se hayan definido previamente. El modelo

es estático, no pueden crearse o borrarse estados, acciones, eventos o transiciones durante la ejecución de la máquina

2.4.1.1 Diagramas de estados

Los diagramas de estado son una *representación gráfica* de las máquinas de estados. En la figura podemos ver un sencillo diagrama de estados.

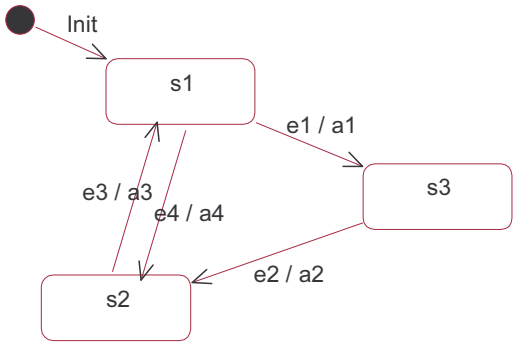


Figura 2.15 Diagrama de estados

Los estados (s1, s2 y s3) se representan por medio de nodos mientras que las transiciones se representan por flechas, las anotaciones que acompañan a éstas representan los pares evento / acción (e1 / a1, e2 / a2, e3 / a3, e4 / a4).

Los diagramas de transición de estados son una técnica muy útil en el análisis y diseño de los sistemas software [CUH99]. Se distingue entre autómatas de Mealy (la salida depende de las entradas y del estado actual) y autómatas de Moore (la salida depende sólo del estado actual).

En la Tesis Doctoral se utiliza la notación de máquinas de estados definida por Harel, o lenguaje de los statecharts, que está incorporada dentro del estándar UML (Unified Modelling Language) [OMG03]. Los statecharts de Harel [Har90] son más claros que las máquinas de transición de estados clásicas de Moore y Mealy, y se usan en dominios tan variados como simulación de vehículos [CKP96], telecomunicaciones [Men99], hypermedia [PMF99] o sistemas de seguridad [Dam99]. Existen otras dos notaciones muy extendidas: la tradicional, generalmente utilizada en los libros científicos, que representa los nodos-estados como círculos; y la utilizada en análisis estructural que los representa como rectángulos.

2.4.1.2 Propiedades

A la hora de realizar el modelado de una máquina de estados hay que tener en cuenta una serie de puntos. Teóricamente, hay que especificar una transición para cada par evento/estado. Si bien, en la realización práctica esto no suele cumplirse; de un modelo en el que no se han definido todos los pares se dice que su *especificación* es *incompleta*. Normalmente sólo los pares de interés se añaden a la especificación, no obstante, a la hora de probar no podemos olvidarnos del resto.

Dos *estados* son *equivalentes* cuando sobre ellos pueden actuar los mismos eventos y, además, desencadenan idénticas acciones. En una máquina de estados bien definida no deberían existir estados equivalentes.

Un *estado* es *alcanzable* cuando existe una transición definida que nos lleva hasta él. Hay varios motivos por los que pueden existir estados inalcanzables:

- *Estados muertos*, cuando una máquina entra en un estado muerto deja de funcionar, por lo que no puede llegarse a ningún otro estado.
- *Bucles muertos*, cuando una máquina entra en un bucle muerto no logra alcanzar ningún estado que no pertenezca a éste.
- *Estados mágicos*, son estados a los que no lleva ninguna transición pero de los que si salen. Son estados iniciales extra. Siempre son debidos a un error.

2.4.1.3 Tablas de transiciones

Cuando el número de estados es demasiado alto los diagramas de estados dejan de ser útiles porque se complican demasiado. Un sistema que supera los 20 estados comienza a ser gráficamente intratable. Una buena alternativa para estos sistemas es representarlos en forma de *tablas de transiciones*. Estas tablas presentan la misma información que los diagramas de estados pero con un formato diferente y pasar de uno a otro no ofrece mayores complicaciones. La máquina de estados de la Figura 2.15 aparece aquí en forma de tablas:

- **Tablas estado – estado**

Las *filas* representan estados aceptados y las *columnas* representan estados resultantes en las *intersecciones* se especifican los eventos y acciones de la transición.

Estado inicial	Estado resultante (evento / acción)		
	s1	s2	s3
s1		e4 / a4	e1 / a1
s2	e3 / a3		
s3		e2 / a2	

Tabla 2.4 Tabla estado-estado

- **Tablas evento – estado.**

Las *filas* representan los eventos las *columnas* contienen los estados en las *intersecciones* se especifican los estados resultantes y las acciones.

Evento	Estado inicial (estado resultante / acción)		
	s1	s2	s3
e1	s3 / a1		
e2			s2 / a2
e3		s1 / a3	
e4	s2 / a4		

Tabla 2.5 Tabla Evento- Estado

2.4.1.4 Limitaciones

Las máquinas de estados tienen dos importantes limitaciones a la hora de utilizarlas para el desarrollo de aplicaciones software.

Por una parte, su *escalabilidad* es limitada. Aunque las herramientas de diseño asistido son una ayuda considerable, leer y dibujar diagramas con más de 20 estados puede resultar bastante engorroso. En este sentido, las tablas de estados nos permiten aumentar el número de estados manejables, pero también tienen sus limitaciones. Generalmente el aumento de estados debe ir acompañado de anidamiento, y esto es algo que este modelo básico no permite.

Por otro lado, este modelo no contempla el hecho de que dos (o más) sucesos independientes *concurran* en el tiempo. La máquina de estados sólo reconoce un estado por instante de tiempo. Una evolución de las máquinas de estados que soluciona estos problemas son los statechart, a los que se dedica el apartado 2.4.2.

2.4.2 Statecharts o Diagramas de Estados Jerárquicos

Una mejora de las máquinas de estados finitos convencionales son los statecharts. Fueron introducidos por Harel [Har87] y su uso se ha extendido mucho en el análisis y diseño de software: están soportados por multitud de herramientas y forman parte del estándar UML. Los statecharts son apropiados para especificar el comportamiento de *sistemas reactivos*, sistemas que se encuentran siempre en un estado y cuyo número de estados posibles es finito. Cuando sucede un *evento*, el sistema reacciona realizando unas determinadas *acciones* y/o cambiando de un *estado* a otro.

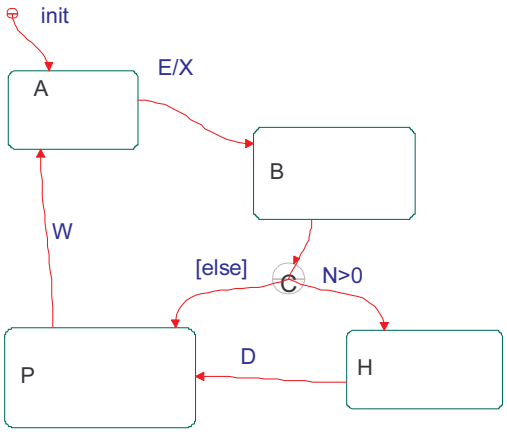


Figura 2.16 Statechart ejemplo

La Figura 2.16 muestra un statechart sencillo. Los estados son los rectángulos de bordes redondeados y están identificados por su nombre. Las flechas que unen los estados son las transiciones. El *estado inicial por defecto* aparece marcado con una flecha con un círculo al principio. E/X es un par *evento/acción*. Esto significa que, en el estado A, cuando se produce el evento E, inmediatamente se ejecuta X y se cambia al estado B. La C rodeada por un círculo simboliza una condición: si N>0, el sistema cambia de B a H, y si no, cambia de B a P. Análogamente, existen los conectores de "switch", que se representan con una S rodeada por un círculo, que sirven para hacer lo mismo con eventos en vez de con condiciones.

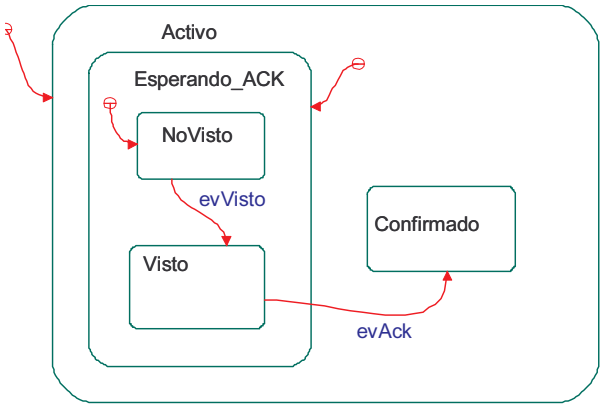


Figura 2.17 Anidamiento de estados

Otra propiedad importante de los statecharts es el anidamiento de estados. En la Figura 2.17 se ve que el estado Esperando_ACK incluye los subestados Visto y NoVisto (estado por defecto). El evento evAck provoca una transición desde el estado Visto al estado Confirmado, pero se ignora en el estado NoVisto. Si la flecha partiera del estado Esperando_ACK, el evento evAck provocaría la misma transición en ambos subestados. La ventaja de esta notación es que aumenta la claridad del modelo, pues se reduce el número de elementos que aparecen.

Un problema de los diagramas de estados convencionales es el de la explosión del número de estados posibles en cuanto el sistema se complica. Por ejemplo, si tenemos un semáforo al que añadimos un timbre y queremos usar para las dos cosas la misma máquina de estados, el número total de estados posibles es el producto de ambos. Para estos casos, los statecharts disponen de los estados AND, que descomponen la máquina de estados en varias máquinas independientes que funcionan paralelamente. La Figura 2.18 es más sencilla que la Figura 2.19, representando ambas lo mismo.

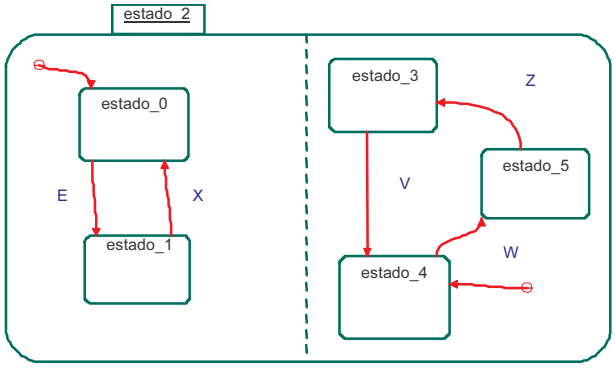


Figura 2.18 Estados AND

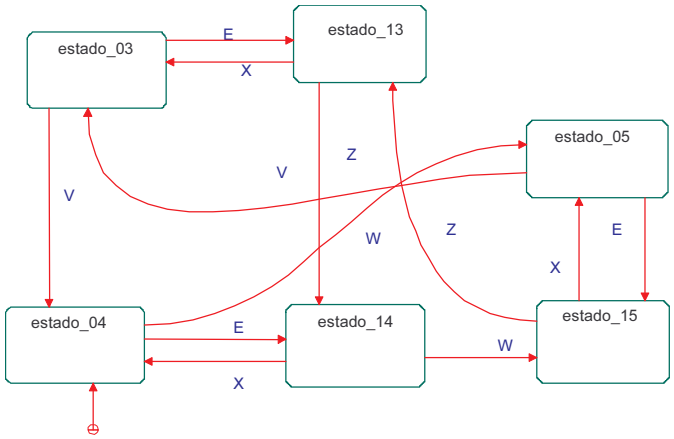


Figura 2.19 Diagrama equivalente sin usar estados AND

Otro elemento importante de los statecharts son los *conectores históricos*, que se representan con una H rodeada por un círculo. En la figura puede verse un ejemplo. Si se vuelve al estado on se pasa al último subestado visitado, y si no se hubiera visitado ningún subestado, se pasa a inactivo.

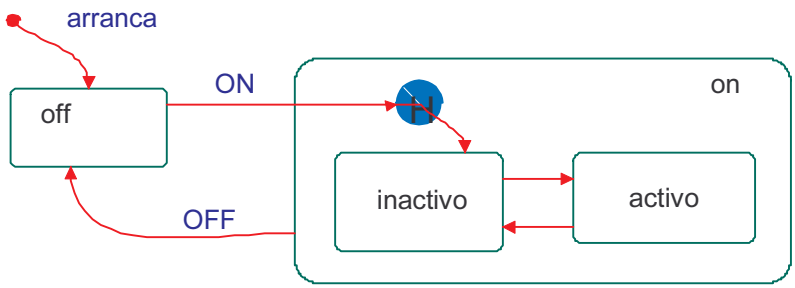


Figura 2.20 Estados Historia

Otros elementos menos usados son las *transiciones compuestas*, que necesitan de la combinación de varios eventos para dispararse. En la Figura 2.21 el cambio del estado A2 puede estar disparado por el evento v o el evento k. También se ve un conector de *unión* de dos transiciones y una transición que se desdobla en dos del estado A a los subestados AND A1 y A2. La T es el símbolo de un estado *término*, al que cuando se llega no hay ya más cambios de estado.

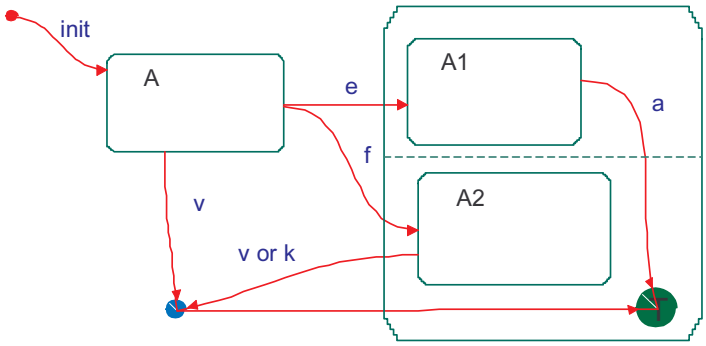


Figura 2.21 Ejemplo de conectores y de estado terminal

Aparte de los elementos gráficos, en la notación de Harel hay elementos textuales, que se describen brevemente a continuación en la Tabla 2.6. Lo expuesto hasta ahora sirve para dar una idea de lo completo y detallado que es el lenguaje de los statecharts para modelar el comportamiento de un sistema.

Expresiones para eventos, condiciones y datos	Expresiones para nombrar (equivalentes a un *define* en C) Son abreviaturas para dar mayor claridad o para ocultar detalles todavía no definidos.
Acciones	Básicas: alteran el valor de los elementos (eventos, condiciones y datos): mandar un evento, hacer que la condición se cumpla o no. Condicionales o iterativas: iguales que en cualquier lenguaje de programación.
Expresiones relacionadas con el tiempo	Timeouts Acciones planificadas para un cierto momento
Reacciones estáticas	Se llama así a las *actividades de entrada* y a las *actividades de salida* de un estado. En el diccionario de datos del modelo aparecen como asociadas al estado. Son tareas que se realizan siempre al entrar o salir del estado. A diferencia de las acciones, que son instantáneas (no pueden interrumpirse) las actividades tienen una cierta duración.

Tabla 2.6 El Lenguaje de Expresión Textual de los Modelos de Harel

2.4.2.1 La Semántica de los statecharts

Harel ha defendido siempre los modelos ejecutables, esto es, que tienen el suficiente nivel de detalle como para poder ser depurados con la ayuda de una herramienta (STATEMATE, Rhapsody, etc.). Véase [HaGe96], [HaGe97] o [HaPo98]. Ahora bien, esa ejecución del modelo necesita una cierta semántica. Por ejemplo, en un sistema real no hay nada instantáneo. Si ele sistema es monoprocesador tampoco hay una concurrencia pura en los estados AND. Hay que saber cómo entiende el sistema la información del statechart. En [HaPo98] y en [HaNa96] con más detalle, Harel expone su visión sobre esta cuestión, que coincide con la estrategia seguida por la herramienta STATEMATE.

[vdB94] recoge diferentes variantes de los statecharts que han aparecido en la literatura. Hay para todos los gustos, pero se pueden dividir entre los que, en función de sus necesidades, simplifican la notación de Harel, como [MLS97], donde se usan los statecharts en combinación con un entorno de verificación automática, [MLSH98] y [Scho97] y los que le añaden nuevos elementos que necesitan por las características del dominio donde aplican la técnica como [Sow98] y [PMF99]. Especialmente interesante es el trabajo de [BGK98], que ordena las transiciones y estados del statechart en función de como afecten a la seguridad del sistema. [Sel93] explica las simplificaciones que hace la metodología

ROOM a los statecharts, que son el origen de los ROOMcharts, para conseguir una ejecución más eficiente.

2.4.2.2 Jerarquías de statecharts

Si un statechart es muy complejo, es posible dividirlo en diferentes partes. Esto tiene las siguientes ventajas:

- Fácil comprensión
- Los usuarios pueden seleccionar las partes que sean relevantes para ellos y desentenderse del resto.
- Puede haber diferentes niveles de detalle.
- Cada parte puede tener su gestión de configuración por separado.
- Cada parte puede diseñarse con una estrategia de diseño propia, la que resulte más adecuada en cada caso (top-down, bottom-up, etc.).
- Modificaciones más fáciles.

La manera de dividir el statechart en varias hojas depende de cada caso. No es bueno tener muchas divisiones fuertemente relacionadas entre sí, porque ello haría perder visibilidad más que ganarla.

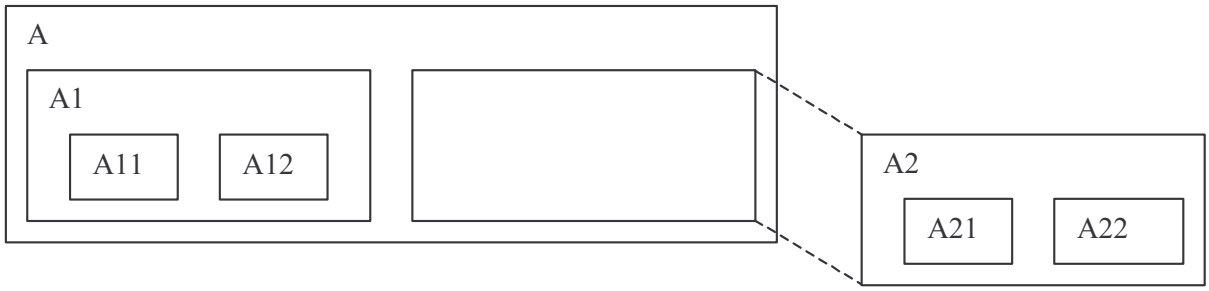


Figura 2.22 Dividiendo un diagrama en hojas

Lógicamente, los flujos salientes y entrantes en cada hoja deben de ser consistentes con los de las hojas restantes. Además, la notación de Harel ofrece los elementos *conectores* que permiten enlazar statecharts: el conector hace referencia a la parte del statechart que no vemos. Los flujos compuestos pueden usarse también para simplificar la representación y sirven para conectar diagramas de actividad y de módulos. Dos diagramas comunes no pueden tener el mismo nombre. Esto sólo esta permitido para los diagramas genéricos, que se explican más adelante.

2.4.2.3 Statecharts Genéricos

Son componentes reutilizables que pueden tener varias instancias dentro del modelo. Las posibles diferencias entre las instancias, tanto en las interfaces externas como en la estructura interna se definen modificando los *parámetros* en el diagrama genérico. Puede haber diagramas genéricos de actividades, de módulos y de estados. La Figura 2.23 muestra un ejemplo sencillo de *definición* e *instancias* de diagramas genéricos

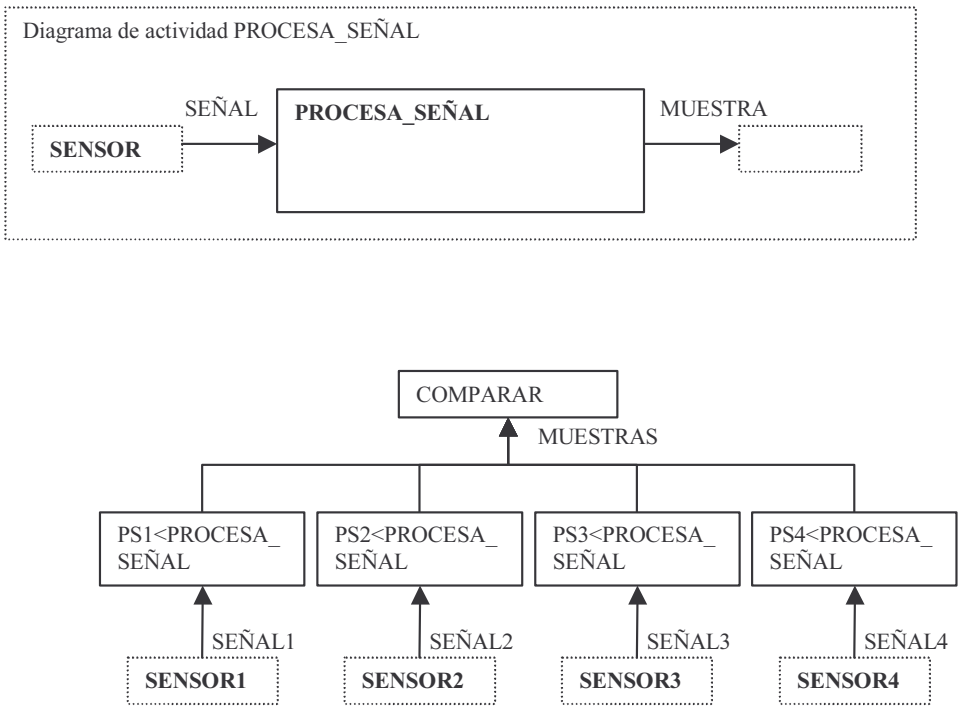


Figura 2.23 Diagrama genérico y sus instancias

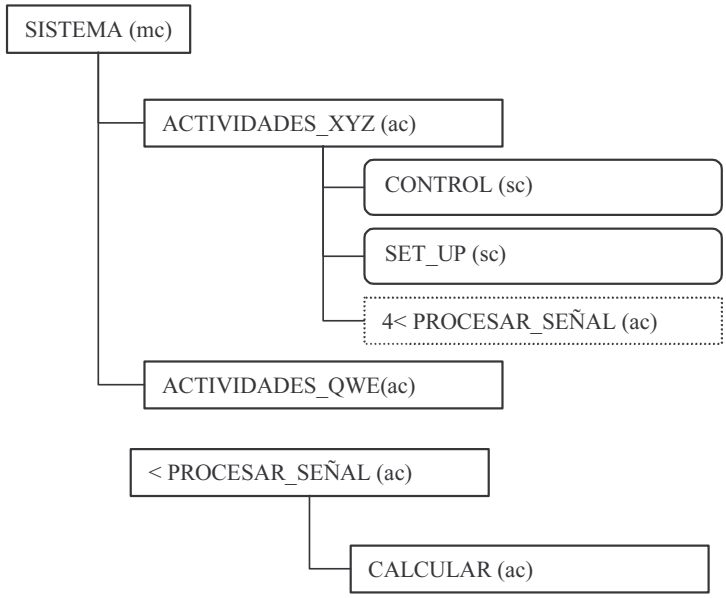


Figura 2.24 Jerarquía de diagramas con charts genéricos

La "caja" de la instancia no puede contener otras "subcajas", ni tampoco otra información de comportamiento que no sea la que hereda. En la Figura 2.24 puede verse cómo se representa un diagrama genérico en la jerarquía de diagramas del modelo.

Cada statechart genérico tiene un conjunto de *parámetros formales*. Estos son *puertos* (canales por los que la información fluye hacia dentro o hacia fuera del componente) o *constantes* (valores para caracterizar una determinada instancia). Cada parámetro formal se describe en el diccionario de datos. Los puertos pueden ser de cualquier tipo de datos o de estructura, también colas.

En el caso de los statecharts genéricos, los parámetros formales del tipo puerto, pueden ser actividades. Esto significa que las instancias de un statechart genérico pueden mandar señales de control a diferentes actividades o comprobar su estado.

En esta línea de buscar máquinas de estados generalizables están los trabajos de [KTW98] y [Wei97], que han desarrollado herramientas que extraen del código en C y C++ máquinas de estados que reemplazan por componentes genéricos, hechos conformes a un patrón de diseño que denomina "autómata de Harel genérico".

2.4.2.4 Los statecharts en UML 2

La versión actual de UML [OMG03], la 2.0, ha introducido los siguientes cambios significativos en la notación de los statecharts:

- Las interfaces pueden tener asociado un statechart de protocolo, que se diferencia de los statecharts normales en que los estados no tienen actividades asociadas.

- Se introducen elementos nuevos en la notación, como los puntos de entrada, los puntos de salida y los terminadores. Los puntos de entrada y de salida se usan si tenemos diagramas de estados anidados dentro de otros diagramas, para mayor claridad (ver Figura 2.25). Los terminadores son una mejora respecto de lo que era el estado final o terminal, pues si se llega al terminador, significa que la entidad asociada al diagrama de estados (se le llama "clasificador" en terminología UML 2 y puede ser una clase, un objeto, etc.) deja de existir en el sistema.

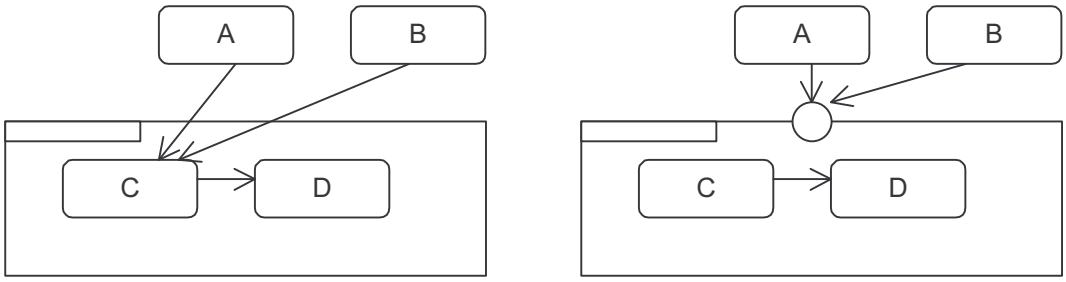


Figura 2.25 Uso de punto de entrada para mayor claridad en el diagrama

- Existe una notación específica para representar statecharts genéricos, representados con una línea discontinua alrededor (ver la figura). Los statecharts derivados de un statechart genérico tienen todo lo que tiene el statechart "padre" y le pueden añadir nuevos estados, nuevas transiciones o nuevos estados "AND". Si algún elemento del statechart "padre" está etiquetado con la palabra clave {final} no puede ser ni extendido ni reemplazado.

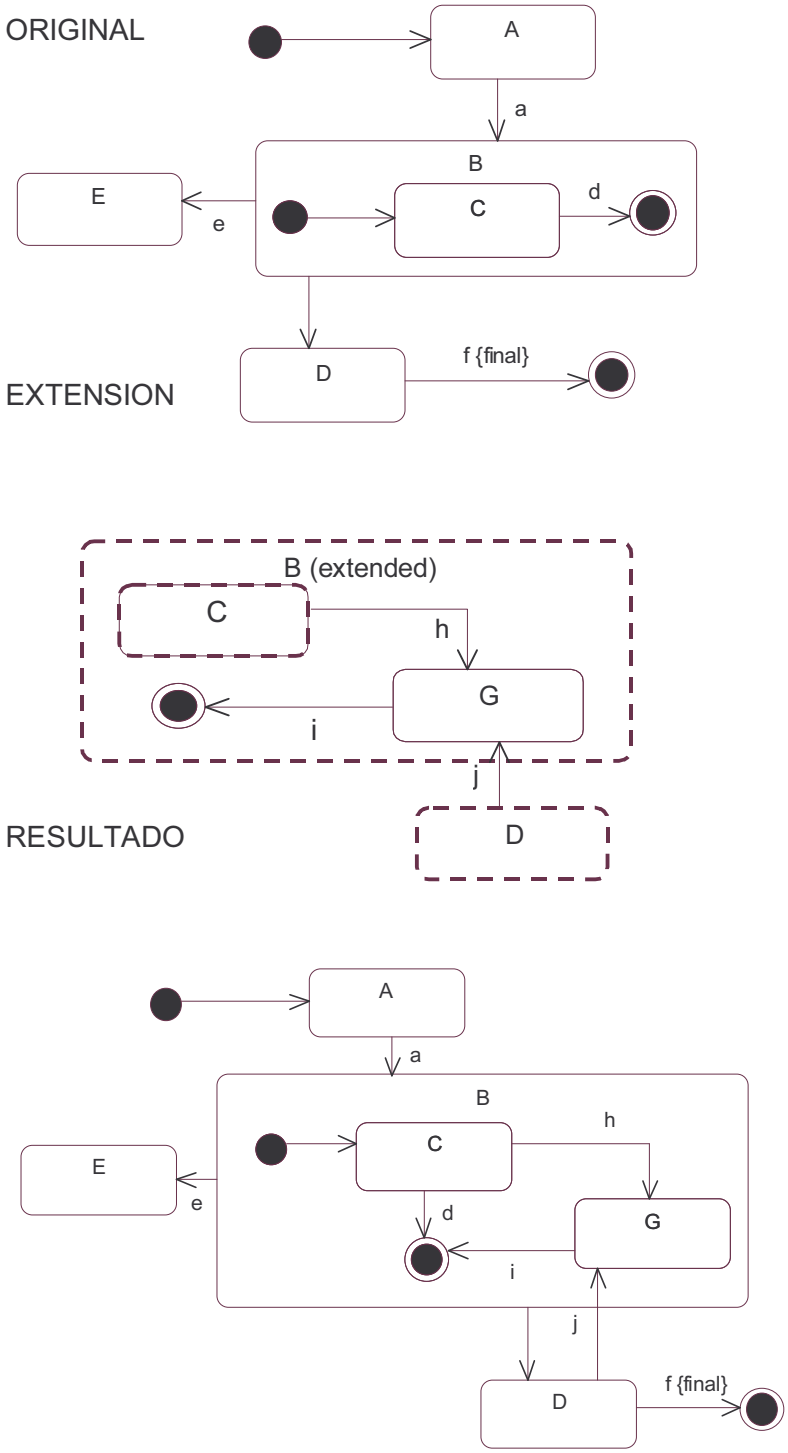


Figura 2.26 Ejemplo de herencia de diagrama de estados en UML 2

2.4.3 Metodologías de análisis y diseño de sistemas de software que usan statecharts

2.4.3.1 OCTOPUS

Introducción

El método OCTOPUS propone un método de desarrollo de software incremental e iterativo orientado a objetos para sistemas empotrados de tiempo real y define un proceso sistemático de desarrollo, fruto de los conocimientos y de la experiencia de sus autores [AKZ96] y [Aw97].

OCTOPUS también puede aplicarse en el desarrollo de sistemas heterogéneos, donde la solución con orientación a objetos es parte de un sistema no orientado a objetos. Por ejemplo, lo habitual es tener un sistema operativo convencional. El problema de mezclar procesos con objetos se resuelve con un procedimiento sistemático y eficiente que asigna objetos a procesos.

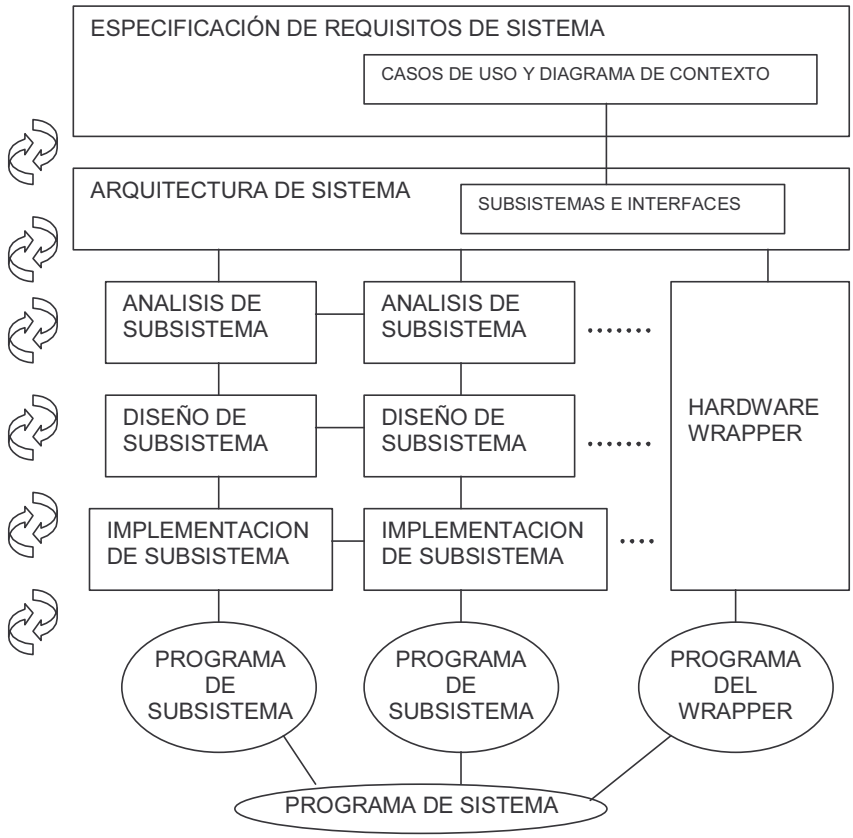


Figura 2.27 Fases del proceso de desarrollo de un sistema software en OCTOPUS

Visión de conjunto del método

La Figura 2.27 muestra las fases del proceso de desarrollo en OCTOPUS. En la fase de especificación de requisitos del sistema se describe la estructura de su entorno mediante un diagrama de contexto, y su funcionalidad con diagramas de casos de uso. El sistema en esta fase es una caja negra.

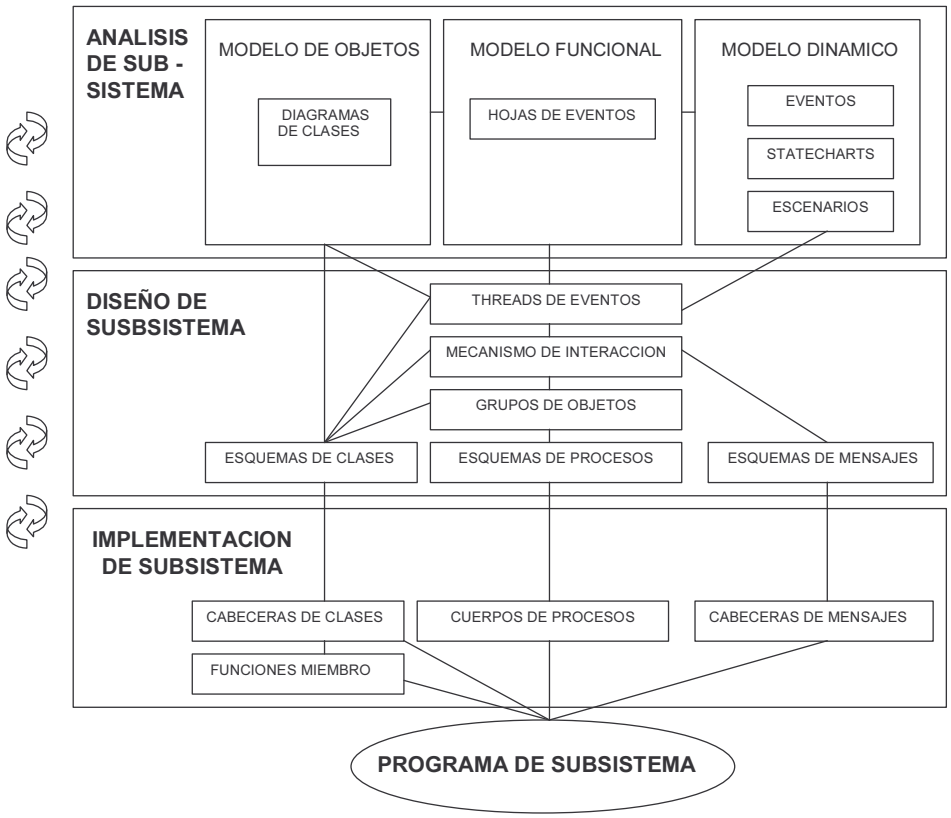


Figura 2.28 Fases del desarrollo de un subsistema en OCTOPUS

La fase de arquitectura del sistema define los subsistemas existentes y las interfaces de cada uno de ellos. Se especifica la funcionalidad de cada uno de los subsistemas y su comportamiento dinámico. El subsistema en esta fase es una caja negra.

El hardware wrapper que aparece en la Figura 2.27 es el elemento que oculta los detalles específicos del hardware al resto del sistema. Es muy importante en los sistemas empotrados, ya que es habitual que utilicen hardware especial. Ofrece una interfaz abstracta de acceso al hardware que es muy importante a la hora de eliminar complejidad en el sistema y aumentar su portabilidad.

OCTOPUS usa la estrategia de desarrollo incremental. Un incremento del sistema está formado por uno o varios subsistemas que pueden probarse conjuntamente porque contienen una funcionalidad determinada. Cada incremento contiene el incremento anterior, y así se suceden los incrementos hasta lograr el sistema final.

La fase de análisis de subsistemas se hace para cada uno de los subsistemas definidos en la fase anterior. En OCTOPUS hay tres modelos definidos, igual que en otros métodos, como OMT:

Modelo de objetos, que define la estructura estática de la aplicación. Describe los objetos del dominio, las relaciones que hay entre ellos (asociaciones, agregaciones, especialización) y los atributos relevantes de los objetos. Su representación gráfica es el diagrama de clases.

Modelo funcional, que describe la interfaz funcional de cada subsistema. No dice nada sobre el comportamiento dinámico del sistema. En OCTOPUS el elemento básico de este modelo son las hojas de operación. Su esquema es el siguiente:

- Nombre de la operación

- Descripción
- Asociaciones: clases y objetos relacionados con la operación
- Precondiciones necesarias para realizar la operación
- Entradas
- Modifica
- Salidas
- Postcondiciones: Qué pasa si la operación tiene éxito/error

Modelo dinámico aborda los aspectos de tiempo real del sistema. Explica bajo que condiciones se hacen las operaciones y en que condiciones pueden realizarse. Tiene varios pasos:

- Análisis de eventos: se trata de determinar qué eventos existen en nuestro sistema.
- Análisis de estados: statecharts, y tablas de acciones
- Escenarios: Muestran en el tiempo cómo interactúan los distintos objetos del sistema entre sí.
- Análisis de estados: statecharts, y tablas de acciones

[AKZ96] define al Statechart como un diagrama de transición de estados convencional + jerarquía + concurrencia + comunicación. El método OCTOPUS usa la notación de Harel, pero con un enfoque propio. A partir de los diferentes eventos presentes en el sistema, se construyen varios statecharts que modelan el comportamiento dinámico de las entidades de del sistema.

Los statecharts se usan para modelar el comportamiento de cada objeto individual. De todas formas, este comportamiento interno es a menudo poco complicado. El comportamiento dependiente del estado se hace más complejo cuando se considera un objeto interactuando con otros objetos.

La estrategia de OCTOPUS es evitar en el análisis una correspondencia estricta entre los statecharts y los objetos. El comportamiento del sistema se refleja en una serie de statecharts, cada uno de los cuales puede contener diferentes grados de detalle. Un statechart puede estar asociado a una clase o a un objeto, pero también puede reflejar aspectos que pertenezcan a varios objetos y clases.

Naturalmente un statechart puede contener una jerarquía de estados anidados y estados concurrentes en cualquiera de los niveles de dicha jerarquía. No es necesario que cada statechart sea tan detallado que se entienda sólo por sí mismo o tan formal y completo que pueda ejecutarse con una herramienta. De esta forma, el análisis ofrece una especificación comprensible del comportamiento del sistema que depende del estado sin necesidad de especificar las interacciones entre los objetos y evitando formalismos innecesarios. Dentro de la fase de diseño, se refinan los statecharts integrándolos dentro de los objetos, y definiendo su tipo (activo o pasivo) y su alcance (compartido, de clase, de objeto).

En esta fase se hacen dos tablas:

- Tabla de statecharts existentes: nombre del statechart, estados elementales y asociaciones
- Tabla de acciones: Estado, Eventos que ocurren en ese estado, Actividades que se ejecutan con ese evento.

El análisis conjunto de eventos y estados da como resultado la tabla de relevancia. Una tabla de relevancia clasifica cada uno de los eventos que puede darse en un cierto estado. Un evento puede ser crítico (el sistema debe de responder al evento siempre y con un tiempo de respuesta máximo), esencial (el sistema no debe de responder al evento obligatoriamente), ignorado o neutral.

Nivel de abstracción	fase de desarrollo	modelo estructural	modelo funcional	modelo dinámico
Sistema (dominio del problema)	Especificación de requisitos del sistema	diagrama de contexto	diagrama de casos de uso del sistema y casos de uso	diagrama de casos de uso del sistema y casos de uso (escenarios)
subsistema	arquitectura del sistema	diagrama de subsistema	modelo funcional del subsistema	modelo dinámico del subsistema
clase	análisis de subsistema	diagramas de clases y tabla de descripción de clases	hojas de operación de subsistemas	lista de eventos del subsistema, grupos de eventos, statecharts, tabla de acciones, tabla de relevancia de eventos y escenarios
objeto	diseño de subsistema	esquemas de clases, esquema de procesos, mensajes entre procesos	esquema de funciones miembro: deben de ajustarse a los esquemas del modelo estructural	"Threads" o hebras de interacción de objetos, "threads" de eventos y grupos de objetos

Tabla 2.7 Proceso de trabajo en OCTOPUS

Fase de Diseño

Durante la fase de análisis, el esfuerzo se concentra en describir las clases existentes en el sistema. La distinción entre objetos y clases se precisa en esta fase. Un objeto de diseño es una instancia de las clases que se han definido en la fase de análisis. El diagrama de objetos muestra los objetos individuales existentes en el sistema y sus relaciones, aunque puede contener clases.

La *interacción de objetos* es el flujo de control y de datos de objeto a objeto. En esta fase se define la forma en que los objetos reciben la notificación de que se ha producido un evento que les afecta y cómo producen la respuesta. La notación de los grafos de interacción entre objetos. Como se ve en la Figura 2.29 es básicamente la misma que en OMT y UML. Los números indican el orden en que se suceden los eventos.

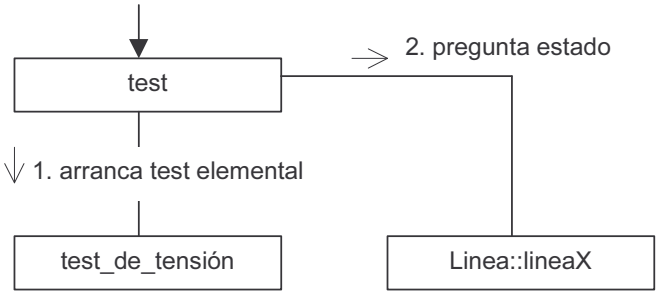


Figura 2.29 Ejemplo de diagrama de interacción de objetos en OCTOPUS

En la fase de análisis se desarrollan algunos statecharts describiendo el comportamiento dependiente

del estado que existe en el sistema. Los statecharts muestran los diferentes estados posibles y los eventos que producen las transiciones de un estado a otro.

En OCTOPUS los statecharts son una herramienta de análisis. No tienen la función de servir como herramienta de prueba de las especificaciones, como hacen el propio Harel [HaPo98] y autores como [Zav96]. Los autores de OCTOPUS afirman que conseguir eso representa un gran esfuerzo para definir con detalle el statechart, con la desventaja de que luego se va tener que rehacer, pues el entorno de simulación del statechart no es idéntico nunca al entorno real del sistema. OCTOPUS permite realizar todos los statecharts que se considere oportuno. Los statecharts se colocan en los grafos de interacción de objetos

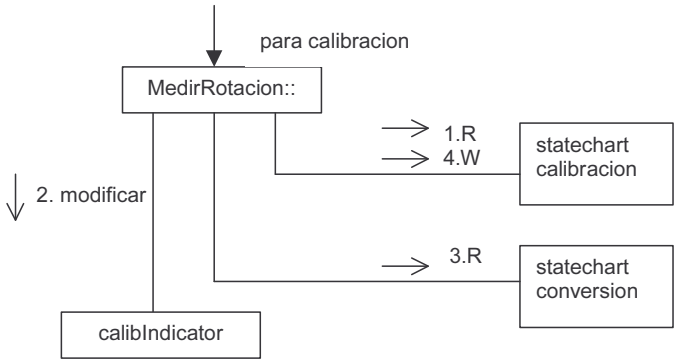


Figura 2.30 Statecharts en el diagrama de interacción de objetos

Un statechart muestra los diferentes estados que puede tener un objeto. Podemos tener:

- Modelo de statechart pasivo. El statechart cambia su estado según le llegan peticiones de otros objetos. Sólo hay flechas de entrada hacia el statechart

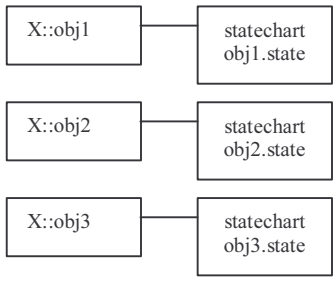
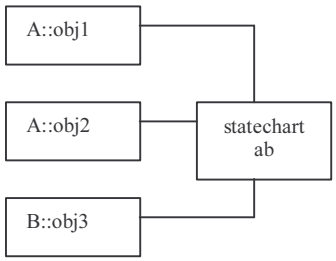
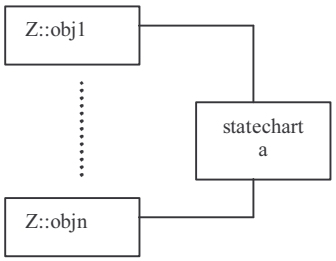
- Modelo de statechart activo. Hace peticiones a otros objetos cuando ocurre un cambio de estado. El statechart tiene entradas y salidas.

Es deseable que un statechart se use por un único objeto. En este caso el statechart es un miembro local de la clase y lo quitamos del grafo de interacción de objetos

Transición del Diseño a la Implementación

Esta fase de OCTOPUS incluye aspectos como interoperabilidad entre distintos lenguajes de programación, control de acceso a las funciones miembro, visibilidad, comunicaciones y rendimiento del sistema. De especial interés son las recomendaciones que se dan para implementar los statecharts.

En [AKZ96] se distinguen dos tipos de statecharts. Si un objeto tiene su propia información de estado, puede declarase como un dato miembro en su clase. El acceso al estado es a través de funciones miembro de la clase. Es el caso A de la tabla. Si varios objetos comparten la misma información de estado, se declara como un dato miembro de tipo “static” en una nueva clase que se introduce para representar el statechart, como en el caso B1 de la tabla. Sólo se instancia un único objeto. Si todos los objetos que comparten esta información de estado se instancian desde la misma clase, los datos que almacenan el estado y las funciones que los acceden se colocan en los objetos. Es el caso B2 de la tabla. En ambos casos, hay que usar un mecanismo de sincronización (semáforos, etc.) para acceder a la información compartida.

Caso	A	B1	B2
Explicación	La información de estado es por objeto	La información de estado es compartida por varios objetos y no todos son de la misma clase	La información de estado es compartida por varios objetos y todos pertenecen a la misma clase
Diagrama			

Esquema de clase	`class X`
	`statechart X state`
	`...`
	`endclass`
	`class Y`
	`statechart Y state`
	`...`
	`endclass`
		`class StatechartAb`
		`static State`
		`...`
		`endclass`
		`class A`
		`statechart ab =newstate`
		`...`
		`endclass`
			`class Z`
			`static statechart A a;`
			`endclass`

Cabecera de clase	`class X`
	`{//...`
	`public:`
	`//member functions`
	`private:`
	`int state;`
	`}`
		`class StateChartAb`
		`{// ...`
		`public:`
		`// member functions`
		`SetState(newstate);`
		`private:`
		`static int state;`
		`}`
		
		`class A`
		`{// ...`
		`StatechartAb::SetState(news tate);}`
			`class Z`
			`{//...`
			`public:`
			`// member functions`
			`private:`
			`static int state;`
			`}`

Tabla 2.8 Diferentes usos de los statecharts en OCTOPUS

OCTOPUS es un método de trabajo que cuenta con el respaldo de haber sido aplicado en proyectos reales en compañías como Nokia y Alcatel. La notación que escogieron en su día los autores fue OMT, pero que bien podría ser hoy UML. El proceso de trabajo que define este método desde la fase de análisis a la implementación es muy completo y sigue teniendo, a mi juicio, plena validez y constituye su aportación más destacada.

2.4.3.2 UML

En el campo del análisis y diseño orientado a objetos, el lenguaje de modelado más extendido es UML (Unified Modelling Language). UML ha sido adoptado como estándar de la OMG (Object Management Group). Ha sido creado por Booch, Jacobson y Rumbaugh, incorporando los elementos de sus respectivas metodologías [Boo96], [Jac94] y [Rum91] y añadiendo aportaciones realizadas por otros autores. El patrocinio de la compañía Rational Corporation ha sido clave en el nacimiento de UML [Rat03]. La versión 0.8 del estándar fue liberada en 1996. En 1997 apareció la versión 1.1, con una participación creciente de personas ajenas a la compañía Rational. En 2003 se ha publicado la versión 2.0, buscando una mayor orientación a la automatización y un mayor grado de abstracción para poder aplicar UML en el desarrollo de sistemas complejos y de tiempo real. Asociado a UML, ha nacido el lenguaje formal OCL (Object Constraint Language), que da mayor precisión a los modelos gráficos de UML, añadiéndoles expresiones de carácter lógico. Para facilitar el intercambio de datos

entre las diferentes herramientas software que soportan UML, se ha definido el estándar XMI (XML metadata interchange), basado a su vez en el lenguaje XML (Extensible Markup Language).

Es oportuno mencionar en este contexto a ROOM, método sobre el que existen numerosas publicaciones a nivel industrial [BoGr96] y académico [MAP97]. La adquisición por Rational Corporation de ObjectTime, principal valedora de la metodología ROOM, aceleró en opinión de [HaPo98] la absorción de este método de análisis y diseño orientado a objetos para tiempo real por UML. Ejemplos de ello son [Sel99] y [Sel98], que incorpora conceptos y notación de ROOM hablando de UML.

UML combina el modelado de datos, modelos con casos de uso, modelado orientado a objetos y modelo de componentes y pretende ser un estándar para visualizar, especificar, desarrollar y documentar sistemas software. Existen varias metodologías de desarrollo que lo utilizan, y puede usarse en diferentes fases del ciclo de desarrollo y con distintas tecnologías de implementación.

UML utiliza las máquinas de estados para especificar el comportamiento. En este apartado se va a dar una visión global del lenguaje, desarrollando la parte correspondiente a los diagramas de estados con más extensión. Una exposición exhaustiva de la notación de UML puede encontrarse en [Dou98] y [Qua98], entre los muchos textos disponibles. Guías rápidas de UML son [Alh99],[Fow99] y [Amb03]. En relación con el tema de la Tesis Doctoral, entre otros, [Fi02] discute la aplicación de UML a sistemas de tiempo real aplicando la herramienta de Rational Rose Real Time, y [Go01] analiza la aplicación de UML al modelado de Líneas de Producto Software. [Jec04] es una explicación detallada de la versión 2.0 del UML.

UML puede usarse para:

- Describir la funcionalidad del sistema con los diagramas de casos de uso y actores.
- Ilustrar las realizaciones de casos de uso con diagramas de interacción.
- Representar la estructura estática del sistema con diagramas de clases.
- Modelar el comportamiento de los objetos con diagramas de transición de estados.
- Explicar cómo tiene lugar a lo largo del tiempo el intercambio de informaciones entre los diferentes objetos con los diagramas de secuencia.
- Detallar la secuencia de operaciones realizadas por medio de los diagramas de actividad.
- Mostrar la arquitectura física de la implementación con los diagramas de módulos y de despliegue.

Diagrama de Casos de Uso

Los casos de uso son descripciones funcionales del sistema. Los diagramas de casos de uso muestran los distintos actores involucrados en cada uno de ellos. Cada caso de uso es una operación completa desarrollada por los actores y por el sistema en un diálogo, y se representa en el diagrama por una elipse. El conjunto de casos de uso representa la totalidad de operaciones desarrolladas por el sistema.

Actor es todo usuario del sistema, que necesita o usa algunos de los casos de uso. Se representa como en la Figura 2.31 y puede ser el usuario final o una entidad externa al sistema (un dispositivo, un sensor, etc.) Para cada caso de uso hay una descripción de lo que el sistema debe de ofrecer al actor cuando se ejecuta el caso de uso. Un ejemplo de aplicación de casos de uso a la especificación de sistemas de tiempo real es [RuH01].

Figura 2.31 Actor y Caso de Uso

Diagrama de clases

Muestra las clases que hay definidas en el sistema y las relaciones que existen entre ellas. Una clase describe un conjunto de objetos con características y comportamiento idéntico. Decir que entre dos clases hay una relación es lo mismo que decir que intercambian información por medio de mensajes. Si dos objetos deben de hablar, tenemos que "tender un cable" entre ellos. Una clase puede estar relacionada con otra de las siguientes maneras:

- Asociación: La clase A usa los servicios de la clase B. Se expresa con una línea entre las dos clases.
- Agregación: La clase A forma parte de la clase B. Se representa con una línea con un rombo vacío en el extremo de la clase propietaria (hay un ejemplo en la Figura 2.32).
- Composición: el propietario es responsable de crear y destruir los objetos Se representa con una línea con un rombo de color negro en el extremo de la clase propietaria.
- Generalización o herencia: La clase B se deriva de la clase A. Se simboliza con una línea terminada en un triángulo, desde la clase derivada o hija a la clase padre. Puede verse también en la Figura 2.32.
- Refinamiento: La clase A es la plantilla que sirve para obtener la clase B. Por ejemplo, si tenemos una clase plantilla para los objetos de tipo pila, y a partir de ella definimos la clase de objetos pila de enteros. Este tipo de relación tiene importancia en el diseño, más que en la fase análisis.

Las clases definen operaciones y atributos, que admiten diferentes grados de visibilidad: públicos o privados. En el diagrama de clases se muestran también la cardinalidad de cada relación (de uno a uno, de uno a muchos, etc.) y su navegabilidad (unidireccional de la clase X a la clase Y o bidireccional).

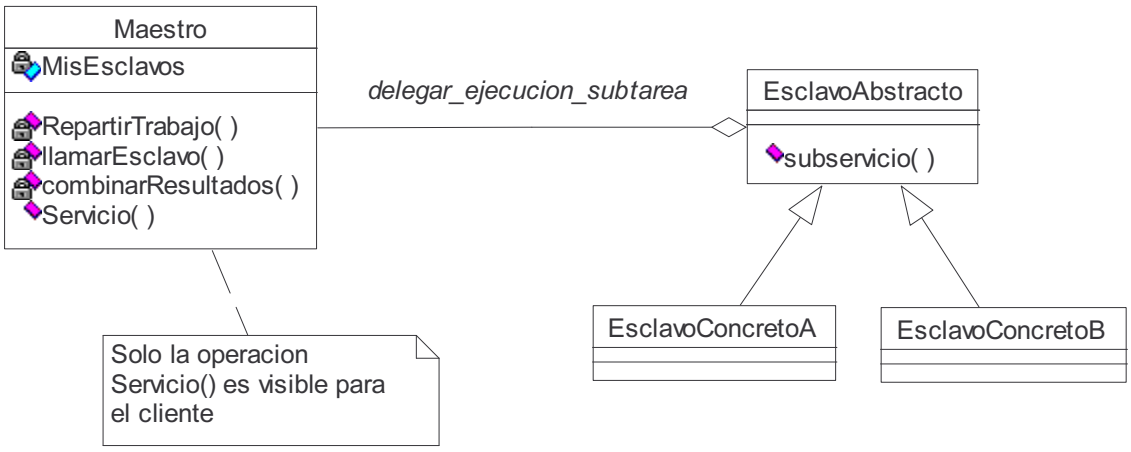


Figura 2.32 Diagrama de clases

Diagrama de colaboración y de interacción

Describen como un grupo de objetos trabaja conjuntamente para realizar una cierta función. Normalmente esta función es un caso de uso. El diagrama muestra en el tiempo la secuencia de intercambio de mensajes entre varios objetos ejemplo.

El diagrama de interacción asocia a cada objeto una línea vertical. Los mensajes que se intercambian entre los objetos se representan con flechas horizontales desde el objeto emisor al objeto receptor (puede haber varios también). El diagrama de interacción puede transformarse en un diagrama de colaboración equivalente. En este diagrama (Ver Figura 2.33) los objetos son cajas rectangulares y los mensajes se representan con flechas a las que va asociado un número que indica el orden en el que se envían los mensajes.

Estos diagramas son de una gran simplicidad y permiten describir de forma muy intuitiva el comportamiento de varios objetos dentro de un caso de uso. No dan información acerca de la funcionalidad que hay que implementar en cada objeto para realizar el intercambio de mensajes, es decir que no se sabe lo que hay "dentro" de las cajas. Para ello, necesitamos recurrir a los diagramas de transición de estados.

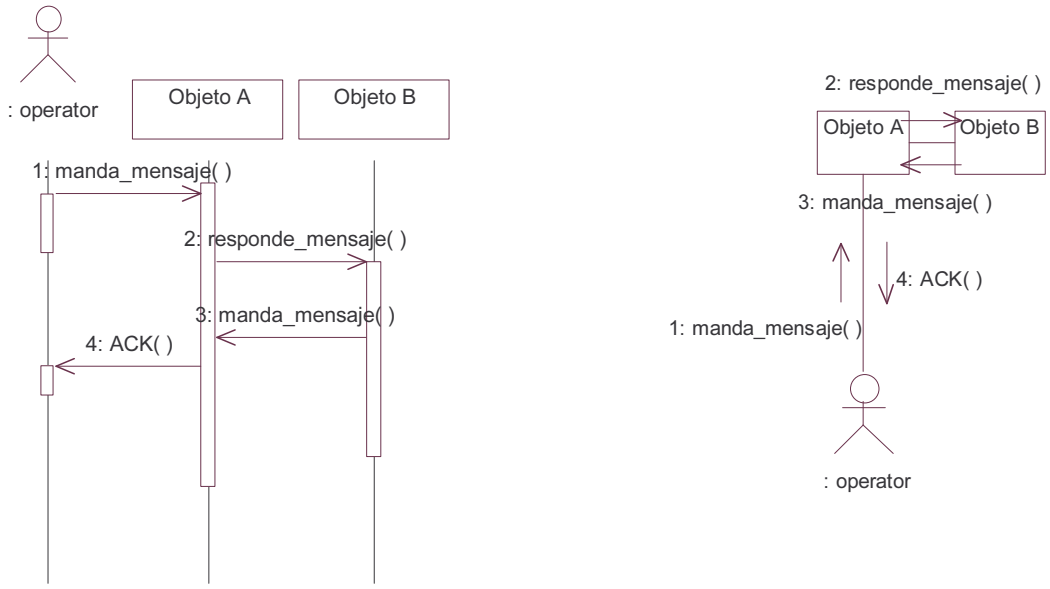


Figura 2.33 Diagramas de colaboración e interacción

Diagramas de actividad

También se han incluido en UML los diagramas de actividad, tomándolos de la metodología de Odell [Ma95]. Estos diagramas muestran los distintos pasos de los que consta una tarea determinada y el orden en que se llevan a cabo y son parecidos a un diagrama de flujo tradicional. Además de los elementos de un diagrama de flujo clásico, como pasos, bifurcaciones, y bucles, incorporan la posibilidad de representar operaciones que suceden en paralelo.

Una de sus ventajas es que no definen los objetos que realizan las operaciones, ni los mensajes que deben de intercambiarse entre ellos, y por eso son adecuados para entender un proceso en su conjunto. Sirven también para modelar procesos que afectan a varios casos de uso.

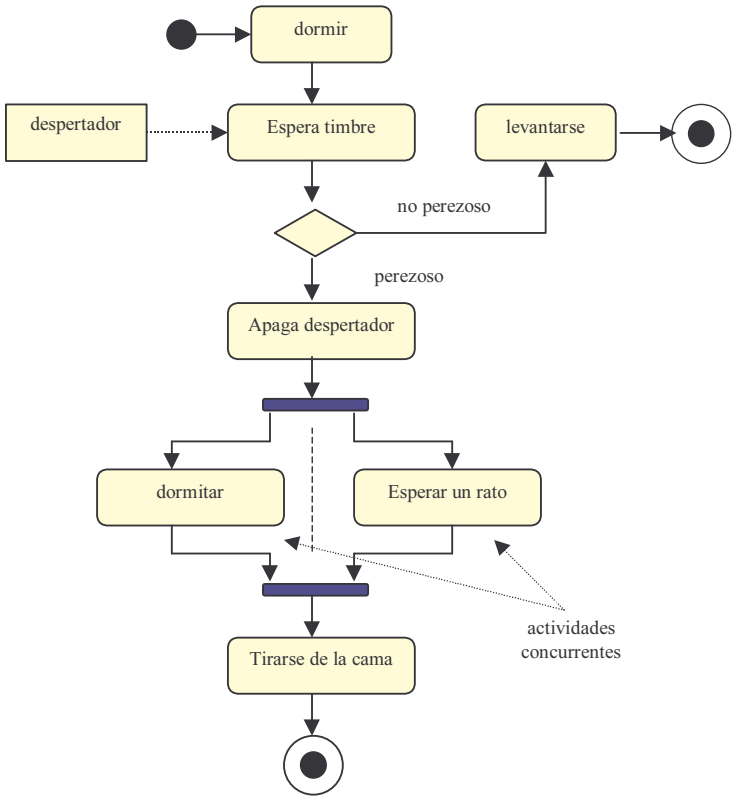


Figura 2.34 Ejemplo de diagrama de actividad

Diagramas de transición de estados

Los diagramas de transición de estados existían mucho antes que comenzara la orientación a objetos. Definen de forma explícita (formal, incluso) el comportamiento de un sistema. Una de sus grandes desventajas es que presuponen que se deben de definir todos los estados posibles de un sistema. Aunque esto es correcto en sistemas pequeños, para sistemas grandes tiene el problema de que el número de estados posibles crece exponencialmente. Para resolver este problema, los métodos orientados a objetos definen para cada clase diagramas de transición de estados por separado. Normalmente, el diagrama de estados de una clase no es excesivamente complejo. Lo que ocurre es que entonces es difícil reconstruir el comportamiento global del sistema a partir de los diagramas de estados de las clases individuales. Los diagramas de interacción y de actividad sirven para reflejar estos aspectos del modelo.

La variante más popular de los diagramas de transición de estados son los statecharts de Harel [HaPo98], que se han explicado ya en el apartado 2.4.2. Fueron adoptados en sus metodologías por Rumbaugh y Booch y finalmente se han incluido en el estándar UML con algunas diferencias en la semántica, que se analizan detalladamente en [Bin00]. La versión de UML 2 ha introducido algunos cambios que se detallan en el apartado 2.4.2.4. Como se ha dicho ya, una de sus cualidades más destacadas es la jerarquía de estados, que hace posible establecer transiciones comunes para todo un grupo de estados. Los procesos instantáneos (que no se pueden interrumpir) se pueden asociar a la entrada en un estado, a la salida de un estado, o a una transición. Los procesos que tienen una cierta duración (que pueden interrumpirse) se llaman actividades. Las transiciones pueden tener una condición asociada, lo que significa que sólo se producen cuando la condición se cumple. El comportamiento de un objeto puede describirse mediante varios diagramas de estados concurrentes.

Los modelos de estados son apropiados para describir el comportamiento de un sólo objeto. Además, son formales y pueden desarrollarse herramientas para verificarlos. Su mayor limitación es que no son buenos para describir el comportamiento conjunto de varios objetos. Para estos casos son más adecuados los diagramas de interacción o los diagramas de actividad. Los diagramas de actividad describen el comportamiento del sistema desde un punto de vista global, que luego debe de implementarse con varios objetos.

Cambios introducidos por UML 2

La versión 2 de UML tiene, como se muestra en la figura, trece tipos de diagramas, de los cuales seis están pensados para modelar la estructura estática del sistema. De los siete diagramas restantes que modelan la dinámica del sistema, hay cuatro de ellos que describen interacciones y que pueden representarse también en forma de tabla si es preciso. Diagramas nuevos respecto de la versión anterior, la 1.5, son los siguientes: diagrama de temporización, diagrama de interacción global y el diagrama de estructura de composición.

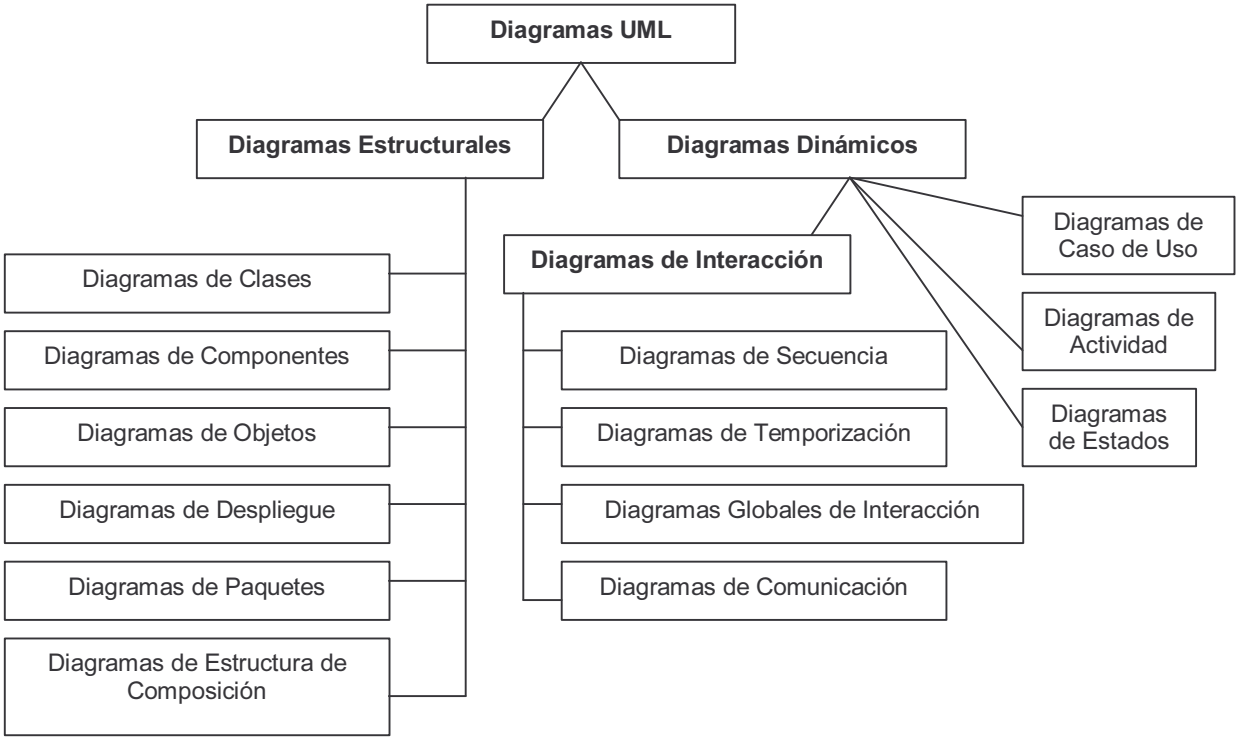


Figura 2.35 Diagramas de UML 2

Las novedades que incorpora UML 2 son las siguientes:

- Los diagramas de actividad son los que han experimentado mayores cambios. El más importante es que dejan de ser una variedad de los diagramas de estados y pasan a basarse en las redes de Petri, y de esta forma se pueden probar mejor y se puede advertir de manera más rápida la ausencia de interbloqueos. También se representa mejor el paralelismo ya que se han eliminado las restricciones existentes en la versión de UML anterior y se han introducido nuevos elementos como por ejemplo el testigo o "token" como base del flujo de ejecución.

- Como alternativa a la representación gráfica del comportamiento dinámico (p.ej. diagramas de estado, diagramas de actividad, etc.) está disponible una representación en forma de tabla, que puede ser útil para la generación de casos de prueba o para comprobar que el diagrama está completo.

- El diagrama de colaboración de UML 1.5 se denomina en UML 2 diagrama de comunicación si bien no se han introducido cambios esenciales. Representa como hasta ahora interacciones durante el ciclo de vida de un objeto, si bien haciendo más hincapié en con qué otros objetos se intercambian mensajes que en la temporización de dicho intercambio

- En los diagramas de clases se eliminan elementos confusos o imprecisos (p.ej. los estereotipos "copy" y "become") pero los elementos esenciales no se han cambiado.

- El diagrama de secuencia, que es el más importante para visualizar las interacciones, se ha hecho en UML 2 jerárquico y divisible, esto es se pueden anidar diagramas de secuencia. Además se han introducido nuevas posibilidades para modificar el flujo de ejecución y representa flujos de ejecución paralelos, lo cual hace posible que se pueda representar con diagramas de secuencia el comportamiento completo y no sólo una selección de escenarios más o menos significativa. Los nuevos diagramas de secuencia de UML 2 incorporan todos los elementos de los ya ampliamente utilizados Message Sequence Charts [ITU96].

- El diagrama de despliegue de UML 2 muestra en la representación de la arquitectura el aspecto del tiempo de ejecución, destacando los aspectos de comunicación entre los diferentes elementos. Se distingue de forma más clara los entornos de ejecución hardware y software y el uso de artefactos (p.ej. especificación de ficheros para generación de código).

- El diagrama de estructura de composición es una novedad introducida en UML 2, y muestra las partes internas de los elementos del sistema y la interacción de las diferentes instancias a través de los canales de comunicación definidos para conseguir un objetivo determinado. Los partidarios de ROOM encontrarán en UML el concepto de puerto en estos diagramas.

- UML 2 introduce también como novedad los diagramas de temporización, ya utilizados ampliamente en electrónica para describir el comportamiento en el tiempo de los objetos.

- El nuevo diagrama de UML 2 de interacción global da una perspectiva de conjunto de los intercambios de mensajes existentes que hace posible construir una estructura jerárquica de tipo top-down con todas las interacciones existentes en ele sistema.

- Otra de las novedades de UML 2 es el diagrama de componentes que muestra las relaciones entre dichos elementos, lo cual hace que UML dé un mejor soporte a algunos estándares del mercado actuales como J2EE y .net

- Los diagramas de estados modelan el comportamiento dinámico del sistema y en UML 2 muestran de forma más clara su relación con los elementos estáticos de modelado (objeto, clases, interfaces, componentes). Para definir de forma detallada las interfaces y los puertos se introducen los diagramas de estados para protocolos.

- El resto de diagramas (casos de uso, objetos y paquetes) no ha sufrido modificaciones.

2.4.3.3 Patrones de diseño para modelar comportamiento basados en estados

En programación orientada a objetos se entiende por patrón una solución probada que se puede aplicar con éxito a un determinado tipo de problemas que aparecen repetidamente en el desarrollo de sistemas software.

Los patrones no son una librería. Van más bien en la línea de un esqueleto básico que cada desarrollador luego adapta a sus necesidades y a las peculiares características de su aplicación. Se describen fundamentalmente en forma textual, acompañada de diagramas y de seudocódigo.

Los patrones de arquitectura son esquemas fundamentales de organización de un sistema software. Especifican una serie de subsistemas y sus responsabilidades respectivas e incluyen las reglas y criterios para organizar las relaciones existentes entre ellos.

Los patrones de diseño son de un nivel de abstracción menor que los patrones de arquitectura. Están por lo tanto más próximos a lo que sería el código fuente final. Su uso no se refleja en la estructura global del sistema.

El interés por los patrones es muy grande actualmente, como lo prueba el hecho de las numerosas y variadas publicaciones que se están haciendo sobre el tema, como [SaCa95], [Sch97], [Sel99]. También se escuchan voces críticas, como se refleja en [Vli98] que, con cierto sentido del humor, habla de los diez principales malentendidos acerca de los patrones.

La referencia fundamental acerca de los patrones es [GHJV95]. Estos autores ya ofrecen el patrón "estado", que da una solución para que el comportamiento de un objeto cambie cuando varía su estado interno. Como se ve en la Figura 2.36, este patrón vale sólo para implementar máquinas de estados sin estados anidados.

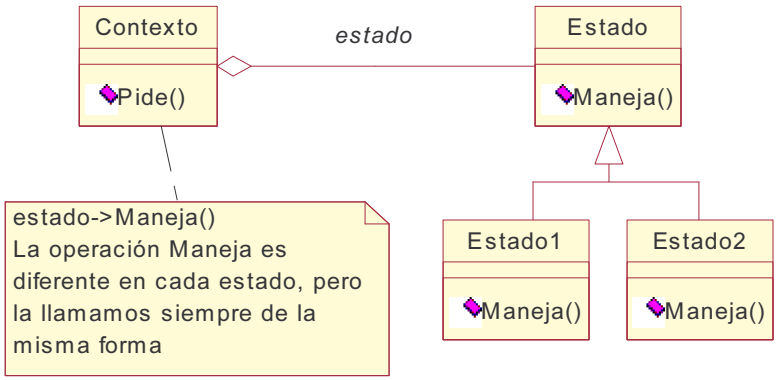


Figura 2.36 Patrón estado

En UML se puede representar que está usándose un patrón mediante una elipse con el borde discontinuo. En [Dou98] y, sobre todo en [Dou99] se contiene una relación de patrones aplicados en sistemas de tiempo real que es una lista de máquinas de estados de carácter general que pueden utilizarse repetidamente en problemas de un determinado contexto.

Como ejemplo representativo, examinemos el patrón "watchdog". En el contexto de los sistemas críticos para la seguridad, es muy importante estar seguro de que determinados componentes están

funcionando correctamente, y por eso se sondean cíclicamente. En caso de que no respondan a tiempo, el sistema debe pasar a un estado seguro de fallo.

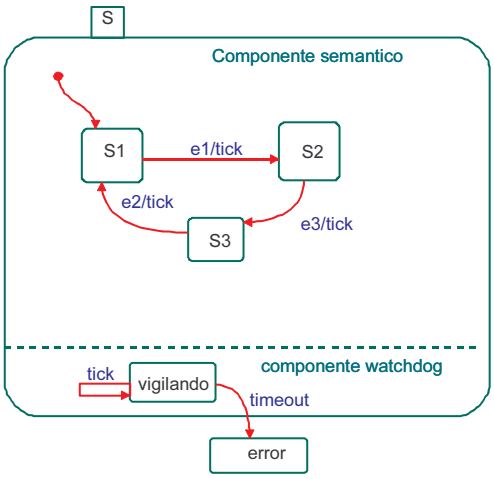


Figura 2.37 Patrón Estado Watchdog

La **solución** es un estado ortogonal de vigilancia (estos es, que coexiste paralelamente con los demás estados sin interferir con ellos). Cuando hay cualquier transición en el componente semántico, se produce la señal de vida que necesita el watchdog para comprobar que el sistema está vivo. Si el watchdog no recibe esa señal en un cierto tiempo, hay un "timeout" y se produce un error.

Como valoración de este patrón [Dou99] dice que si la CPU para el componente watchdog es común con la del componente semántico, en caso de que se cuelgue, no es posible llegar al estado de error. La solución, según [Dou99], es usar un reloj y una CPU independiente para el watchdog, pero eso no es siempre posible.

2.4.4 Otros métodos que usan diagramas de estados

2.4.4.1 VFSM (Virtual Finite State Machine)

VFSM es una técnica nacida en la compañía Bell que permite especificar el comportamiento de un módulo de software como una máquina de estados finitos. Esta técnica definida por sus autores en [God96] como "un paradigma de diseño en el cual el comportamiento de un módulo se especifica con un autómata; y un paradigma de implementación que consiste en una estructura de diseño que define la interfaz entre la especificación del control y el resto de la implementación". Con la ayuda de herramientas, se afirma en [Flo97] y [Ard96] que se puede probar, generar el código y hacer documentación del sistema. La Figura 2.38 muestra un ejemplo sencillo de una VFSM tomado de [Flo97]. VFSM añade únicamente a las máquinas de estados finitos convencionales el uso de variables booleanas.

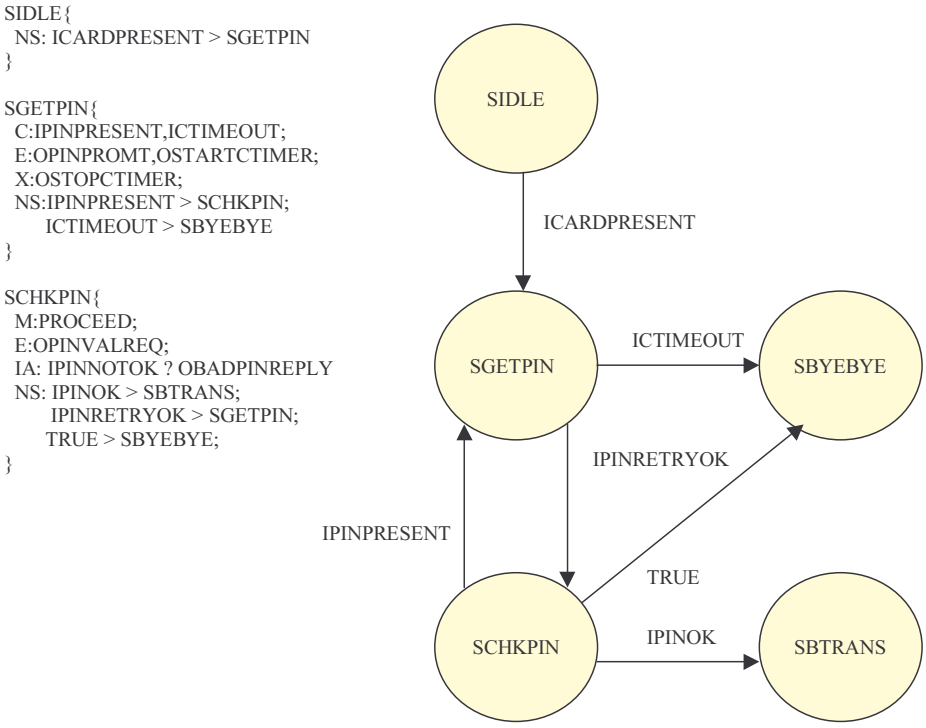


Figura 2.38 Ejemplo de especificación de una VFSM y diagrama de transición de estados

Los autores de VFSM han realizado estudios comparativos de su método con otras técnicas de especificación. El más exhaustivo, por el número de métodos cubiertos, es [Ard96]. En [Flo97] afirman que en comparación con SDL y los statecharts de Harel, que se usan más según ellos en la definición de la arquitectura del sistema, VFSM se usa en el diseño de bajo nivel. La otra diferencia que ponen de relieve es la capacidad expresiva de SDL y los statecharts es mucho mayor que la de las VFSM y eso tiene el peligro, según ellos, de que pueden usarse incorrectamente como un lenguaje de programación, que mezcla el diseño con la implementación y que no puede validarse ni simularse fácilmente. VFSM, en cambio, es mucho más simple y obliga a hacer una especificación menos detallada que es más fácil de entender. Pero al mismo tiempo, afirman que han podido capturar la mayoría de las aplicaciones que han encontrado.

2.4.4.2 Análisis estructurado: método de Ward y Mellor

Es una variante al análisis estructurado de De Marco, que incluye también elementos de control, y que se ha quedado ya un tanto obsoleto. Los diagramas de este método, llamados *esquemas de transformación* contienen nodos de *datos* y nodos de *control*. Los nodos están unidos por flechas que expresan flujo de datos en el caso de ser flechas continuas y flujo de control en el caso de las flechas discontinuas. El diagrama principal es el diagrama de contexto, que muestra las interacciones del sistema con su entorno. Los nodos (también llamados *transformaciones*) de control contienen una máquina de estados.

Puesto que este método usa los diagramas de estados convencionales, no tiene las ventajas adicionales que proporciona la notación de los statecharts, especialmente las jerarquías de estados, concurrencia o estados historia. Tampoco existen mecanismos para manejar componentes similares dentro de un modelo, como los statecharts genéricos.

2.4.4.3 Structured Description Language (SDL)

SDL es un lenguaje de especificación de comportamiento para sistemas de tiempo real definido por la ITU [SDL95] y que goza de gran aceptación en el ámbito de los sistemas de telecomunicación Se puede aplicar en varias fases del ciclo de vida: especificación de requisitos, diseño detallado del sistema y descripción de casos de prueba. El comportamiento del sistema se modela con un conjunto de máquinas de estados finitos conectadas entre sí. Estas máquinas de estados tienen *estados*, *entradas*, *salidas* y *transiciones*. Un estado en SDL no puede englobar dentro de él otros estados. Tiene una notación textual y otra notación gráfica. En SDL, el elemento básico de un modelo es el *proceso*. Un proceso es una máquina de estados que a su vez puede comunicarse con otros procesos. Se pueden hacer los modelos en SDL más compactos y fáciles de entender usando *paquetes* como los de UML y otros mecanismos parecidos.

A continuación se muestra un ejemplo tomado de [SDL95] de la especificación de un sencillo proceso, en forma textual y en forma gráfica, que controla un juego y que puede arrancar el juego, terminarlo, guardar los puntos y comunicárselos a los usuarios.

```
service Controla_Juego;
/*Contador que guarda los puntos*/
    dcl CONTADOR Integer,
    A Integer; /* incremento contador */

    start;
         output SUBSCR;
         output JUEGOID to JUGADOR;
         task CONTADOR:=0;
         nextstate comienza;

    state comienza;
         priority input PRUEBA_RESULTADO(A);
         task CONTADOR:=CONTADOR+A;
         nextstate -;
         input RESULTADO;
         output PUNTOS(CONTADOR) to JUGADOR;
         nextstate -;
         input FINJUEGO;
         output ACABA_JUEGO;
         stop;
    endstate comienza;
endservice Controla_Juego;
```

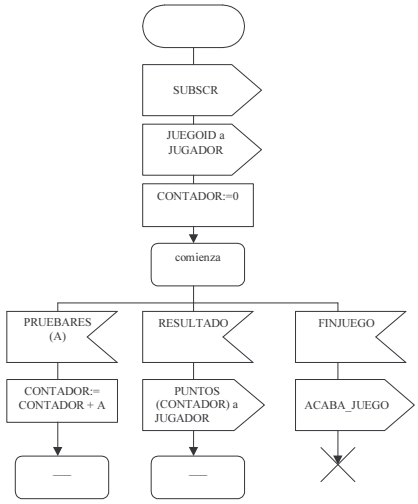


Figura 2.39 Ejemplo de un diagrama de proceso en SDL

El propio Harel en [HaPo98] admite que el poder expresivo de SDL y los statecharts es muy parecido. Ambos métodos pueden aplicarse en diseño estructurado y en diseño orientado a objetos.

[Ek95] comenta que el punto fuerte de los métodos de diseño orientado a objetos es su carácter intuitivo que facilita enormemente la fase inicial de análisis de un sistema. SDL, sin embargo, define un sistema con tanto detalle, que es muy fácil generar los casos de prueba del sistema a partir de los modelos SDL. Unos casos de prueba además, que son automatizables. La herramienta Tau Tester, de la casa Telelogic [Tel02], ofrece la posibilidad de convertir statecharts en SDL y viceversa, y, a partir de la especificación en SDL, generar casos de prueba de acuerdo a la notación TTCN (Tree and Tabular Combined Notation), estándar ISO 9646-3 o ITU X.292 [ITU92].

2.4.5 Pruebas de Diagramas de Estados

Los diagramas de estados son ventajosos no sólo a la hora del análisis y el diseño, sino también en la fase de pruebas del software. Recorriendo de forma sistemática todos los estados y transiciones existentes, estamos seguros de haber verificado completamente el sistema. Este proceso puede hacerse

de forma manual, o si el modelo a probar es muy complejo, de forma automática. Para una aproximación general al desarrollo de pruebas basadas en diagramas de estados se puede consultar [Bei95] y [Bin00].

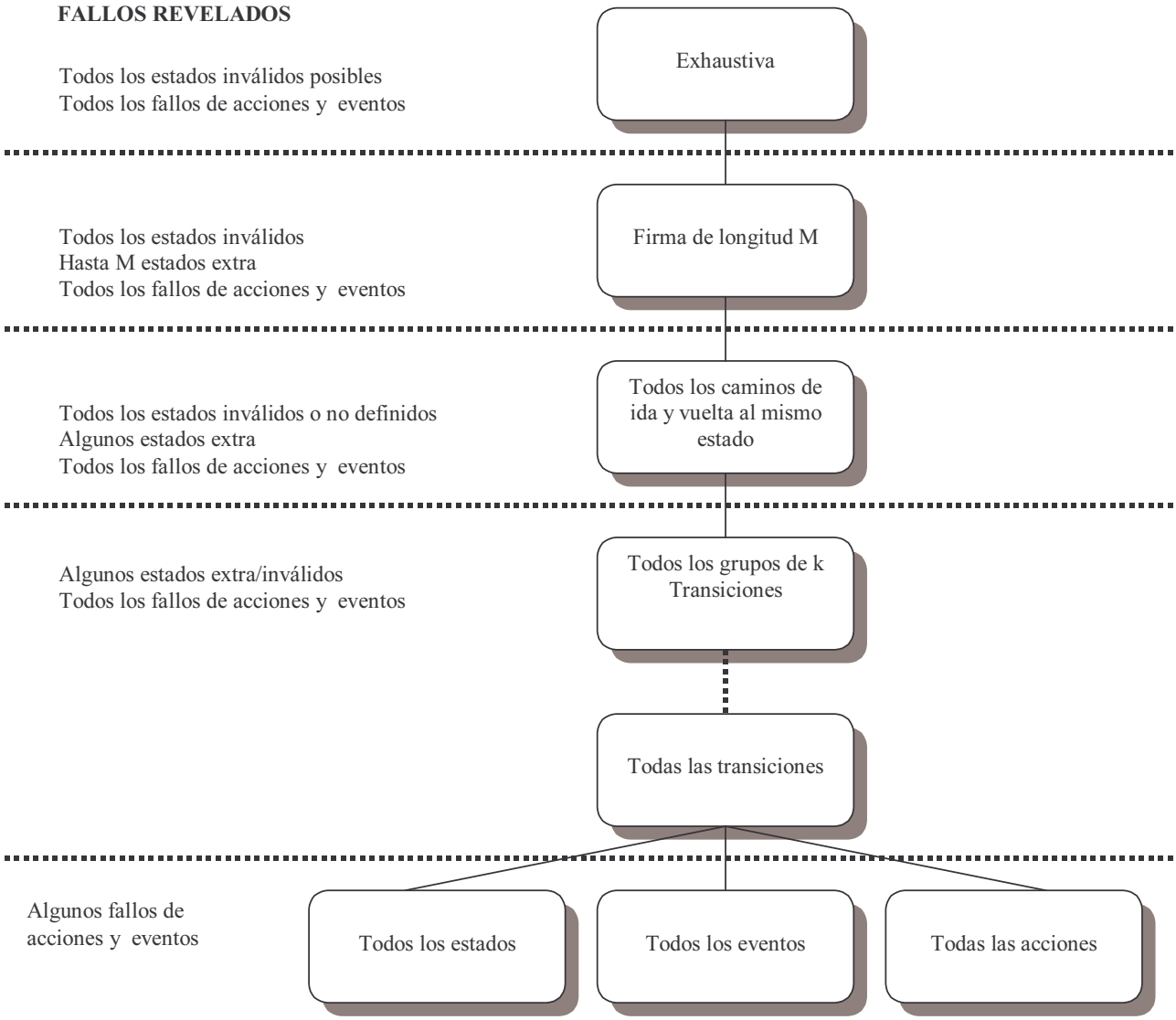


Figura 2.40 Estrategias de prueba de máquinas de estados, ordenadas por potencia

2.4.5.1 Modelo de fallos

Los posibles problemas que puede tener una máquina de estados son los siguientes:

- Una transición falta o es incorrecta (el estado resultante no es correcto, pero es un estado definido).
- Un evento falta o es incorrecto (se ignora un mensaje válido).
- Una acción falta o es incorrecta (una transición produce un efecto no deseado).
- Un estado falta, sobra o está corrupto (valor erróneo con consecuencias impredecibles).
- Un camino fantasma (se acepta un evento indebidamente).

- Un mensaje ilegal produce un error.
- Una puerta trasera (se aceptan mensajes no definidos).

Estos fallos pueden darse aislados o de forma simultánea. El número y tipo de los fallos que vamos a encontrar está en función de la manera en cómo se va a probar: la estrategia de prueba.

2.4.5.2 Estrategias de prueba

Hay muchas estrategias posibles de generación de casos de prueba a partir de un diagrama de estados. En la Figura 2.40, tomada de [Bin00], se muestra una clasificación de diferentes estrategias de pruebas, ordenadas de mayor a menor grado de exhaustividad. La más exhaustiva de todas (parte superior de la gráfica) no es viable por el esfuerzo que implica y la menos exigente (parte inferior) no revela apenas fallos.

El coste de las pruebas basadas en diagramas de estados debe medirse según dos parámetros: el tiempo y los recursos necesarios para cada caso de prueba y, dentro de cada caso, para cada mensaje (evento) del mismo. El coste de diseño, ejecución y análisis posterior de un caso de prueba rara vez depende del número de sus eventos; el coste de los mismos se ve reflejado en el almacenamiento y ejecución de los mensajes así como en su desarrollo. El tiempo de ejecución y el espacio de almacenaje suelen incrementarse proporcionalmente con el tamaño del modelo a probar; este tamaño puede dimensionarse con dos parámetros: **n**- *número de estados* y **k**- *número de eventos*.

Por piezas

Esta es la estrategia menos exigente, se van probando las especificaciones por conjuntos: todos los estados por una parte, todos los eventos por otra y por último todas las acciones. Al no involucrarse directamente en la estructura de la máquina de estados los fallos de comportamiento se detectan sólo de manera casual. Es posible pasar por todos los estados sin que tengan lugar todos los eventos, de igual manera que es posible ejecutar todos los eventos sin necesidad de pasar por todos los estados o producir todas las acciones o, incluso, provocar todas las acciones sin visitar todos los estados ni aceptar todos los eventos.

Son muchos los fallos que escapan a esta estrategia, por lo que no es aconsejable si se quiere realizar unas pruebas serias. Por un esfuerzo ligeramente superior se puede pasar a una estrategia del tipo ‘todas las transiciones’, que detecta un abanico mayor de errores, como se recoge en la tabla de la Tabla 2.9.

Todas las transiciones

Se dice de una estrategia que tiene cobertura para todas las transiciones si cada transición se realiza al menos una vez. Para ello es necesario pasar por todos los estados, generar todos los eventos y ejecutar todas las acciones, sin importar el orden con que se haga.

Esta estrategia de prueba consiste en generar al menos un mensaje de prueba por cada transición del modelo. El número de casos de prueba es igual al número de transiciones, que, según la máquina, puede llegar a ser muy elevado.

De esta manera se descubren un alto número de errores a costa de una complejidad proporcional al tamaño de la máquina. Puesto que se requiere una transición en cada estado obtenemos un total de k.n casos de prueba, este es también el número de mensajes que se obtienen, supuesto que se genere un mensaje por transición.

Con esta estrategia está garantizado que todos los pares evento/acción incorrectos o perdidos van a ser detectados; sin embargo, no muestra si ha pasado por estados incorrectos. Sólo si la máquina está completamente especificada detecta caminos fantasma.

Todos los grupos de k transiciones

Este tipo de estrategias utiliza secuencias de transiciones específicas de n eventos que se ejecutan al menos una vez. A estas secuencias se las conoce con el nombre de *switch*. Una ***x-switch*** designa una secuencia de x-1 transiciones; de forma que 0-switch indicaría una transición, 1-switch una pareja, … Una estrategia del tipo 'todas las transiciones' como la vista en el apartado anterior sería equivalente a una 1-switch. A estas secuencias se las conoce también con el nombre ***k-tuplas***, una 2-tupla incluye todos los pares (k=2), una 3-tupla todas las tripletas (k=3), …

Todos los caminos de ida y vuelta al mismo estado

En este tipo de estrategias se definen secuencias de transiciones específicas de manera que el estado final de la secuencia coincide con el inicial. Estas secuencias se ejecutan al menos una vez. La secuencia mas corta es aquella de una sola transición, la longitud de la más larga depende de la máquina en cuestión.

Al conjunto de secuencias circulares se lo conoce con el nombre de ***n-switch***, e incluye todas las secuencias de longitud menor o igual a n. Un conjunto de pruebas que utiliza el mayor n-switch posible revela todas las parejas evento/acción erróneas. También se detectan parte de los estados erróneos, pero no garantiza que se descubran los fallos de implementación con más de n estados incorrectos.

Estrategia N+

La generación de los casos de prueba se lleva a cabo en cuatro pasos:

- Generación de una tabla de transiciones.
- Generación del árbol de transiciones con la información de la tabla de transiciones.
- Generación de la prueba de recorrido de la máquina de estados.
- Generación de la prueba de caminos ilegales.

Se realizan así dos comprobaciones:

- Recorrido de la máquina de estados: una secuencia de eventos permitida produce el estado final deseado.
- Detección de caminos ilegales: Se da un valor al estado y se producen todos los eventos ilegales, para asegurar que el estado no toma un valor erróneo. Es más sencillo cuando la propia implementación ofrece una función para acceder al valor del estado.

Para hacer la prueba de recorrido de la máquina de estados, se construye un *árbol de transiciones* siguiendo este procedimiento [Cho78]:

1. El estado inicial es el nodo raíz del árbol (los demás son nodos hoja). Si hay varios estados iniciales posibles, se define un estado previo a todos ellos al que se llama estado alfa.

2. Examinar el estado que corresponde a cada nodo hoja no terminal en el árbol y las transiciones que parten de este estado. Se dibuja como mínimo un nuevo nodo por cada una de estas transiciones. Cada nuevo nodo representa un evento y un estado resultado de una transición.

- Si la transición no tiene condiciones que la protejan, se dibuja una nueva rama.
- Si la condición que protege la transición es un predicado simple o compuesto sólo por operadores AND, se dibuja una nueva rama.
- Si la condición que protege la transición es un predicado compuesto por operadores OR, se dibuja una nueva rama por cada combinación de valores que hacen que se cumpla la condición.

3. Para cada flecha y cada nodo dibujados en el paso 2:

4. Anotar para cada rama su evento, su condición (en el caso de que la tenga), y su acción. Si el estado que representa el nuevo nodo está ya representado en el árbol o es un estado final, este nodo es un nodo terminal (no se dibujan más transiciones en este nodo).

5. Repetir los pasos 2 y 3 hasta que todos los nodos hoja sean terminales.

El árbol de transiciones para la Figura 2.19 está en la Figura 2.41, con los nodos terminales marcados con el borde más grueso.

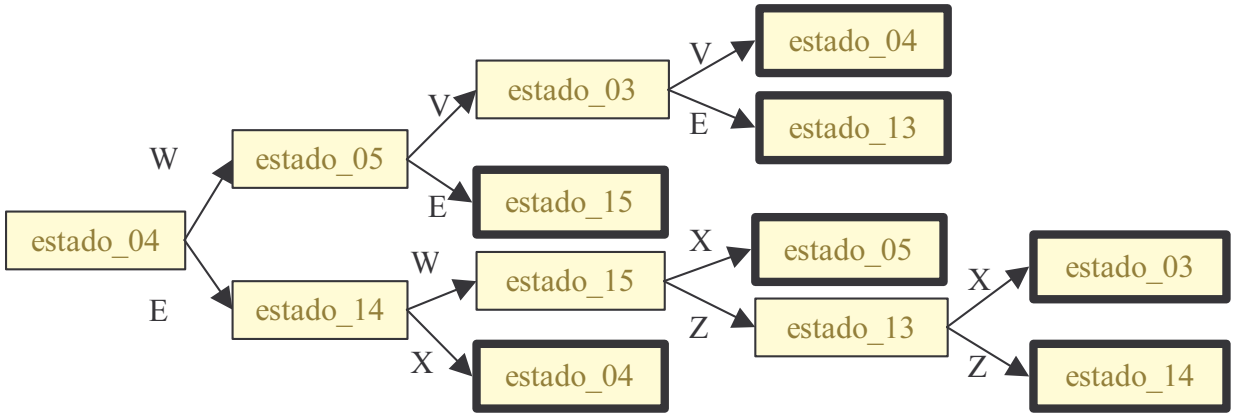


Figura 2.41 Arbol de transiciones

La siguiente tabla contiene todos los casos de prueba del árbol de transiciones. Cada camino desde el nodo raíz a un nodo terminal es un caso de prueba. Es lo que se ha llamado prueba de recorrido de la máquina de estados anteriormente: asegura que la implementación se comporta según el modelo definido mediante la máquina de estados.

ID	**Valores de Entrada**		**Resultado esperado**	
	Evento	Condición	Acción	Estado
1.1	W	--	--	estado_05
1.2	E	--	--	estado_15
2.1	W	--	--	estado_05
2.2	V	--	--	estado_03
2.3	V	--	--	estado_04
3.1	W	--	--	estado_05
3.2	V	--	--	estado_03
3.3	E	--	--	estado_13
4.1	E	--	--	estado_14
4.2	W	--	--	estado_15

ID	Valores de Entrada		Resultado esperado	
	Evento	Condición	Acción	Estado
4.3	X	--	--	estado_05
5.1	E	--	--	estado_14
5.2	W	--	--	estado_15
5.3	Z	--	--	estado_13
5.4	X	--	--	estado_03
6.1	E	--	--	estado_14
6.2	W	--	--	estado_15
6.3	Z	--	--	estado_13
6.4	Z	--	--	estado_14
7.1	E	--	--	estado_14
7.2	X	--	--	estado_04

La prueba de caminos ilegales asegura que no hay reacciones inesperadas ante eventos ilegales, del tipo "pulso en mi base de datos bancaria Control-Shift-X-W y aparece el buscaminas en la pantalla". Para ello, se da al estado todos los valores posibles, se genera para cada valor todos los posibles eventos y se ve qué es lo que sucede. En la tabla del ejemplo se usan las secuencias de eventos que se han probado con la tabla anterior para inicializar el estado. En el campo resultado esperado se ha puesto excepción de evento ilegal, pero se puede definir otra diferente (parada de sistema, etc.).

ID	Caso de Prueba			Resultado esperado
	Inicialización	Estado	Evento	
10.0	Estado por defecto	estado_04	V	Excepción Evento ilegal
11.0	Estado por defecto	estado_04	X	Excepción Evento ilegal
12.0	Estado por defecto	estado_04	Z	Excepción Evento ilegal
13.0	4.1, 4.2, 4.3	estado_05	W	Excepción Evento ilegal
14.0	4.1, 4.2, 4.3	estado_05	X	Excepción Evento ilegal
15.0	4.1, 4.2, 4.3	estado_05	Z	Excepción Evento ilegal
16.0	2.1, 2.2	estado_03	W	Excepción Evento ilegal
17.0	2.1, 2.2	estado_03	X	Excepción Evento ilegal
18.0	2.1, 2.2	estado_03	Z	Excepción Evento ilegal
19.0	3.1, 3.2, 3.3	estado_13	E	Excepción Evento ilegal
20.0	3.1, 3.2, 3.3	estado_13	W	Excepción Evento ilegal
21.0	3.1, 3.2, 3.3	estado_13	V	Excepción Evento ilegal
22.0	7.1	estado_14	V	Excepción Evento ilegal
23.0	7.1	estado_14	E	Excepción Evento ilegal
24.0	7.1	estado_14	Z	Excepción Evento ilegal
25.0	1.1, 1.2	estado_15	W	Excepción Evento ilegal
26.0	1.1, 1.2	estado_15	V	Excepción Evento ilegal
27.0	1.1, 1.2	estado_15	E	Excepción Evento ilegal

Firma de longitud M

La firma de estados de longitud M es una técnica que ve a la implementación que se prueba como una caja negra cuyo estado interno no es accesible directamente. La "firma" de estados es una secuencia de acciones de salida que son únicas para un estado de partida determinado. En la especificación de requisitos del sistema está definida una secuencia de eventos que produce la "firma". Para comprobar si se alcanza un estado se ejecuta la secuencia de eventos seleccionada y se comprueba que la firma es la esperada, de no serlo, se asume que se ha llegado a un estado corrupto.

La siguiente tabla ofrece una comparativa de las distintas estrategias.

Errores:	Todos los eventos	Todos los estados	Todas las acciones	Todas las transiciones	N+
Transiciones perdidas	(1)				SI
Caminos fantasmas				(2)	SI
Acciones incorrectas				SI	SI
Estados perdidos		SI		SI	SI
Estados corruptos				SI	SI

Tabla 2.9 Comparación de las estrategias de prueba observables

(1)Sólo si la relación evento/transición es 1 a 1.
(2)Sólo si el modelo está completamente especificado.

2.4.6 Escenarios

Describen como un grupo de objetos trabaja conjuntamente para realizar una cierta función mostrando en el tiempo la secuencia de intercambio de mensajes entre ellos.

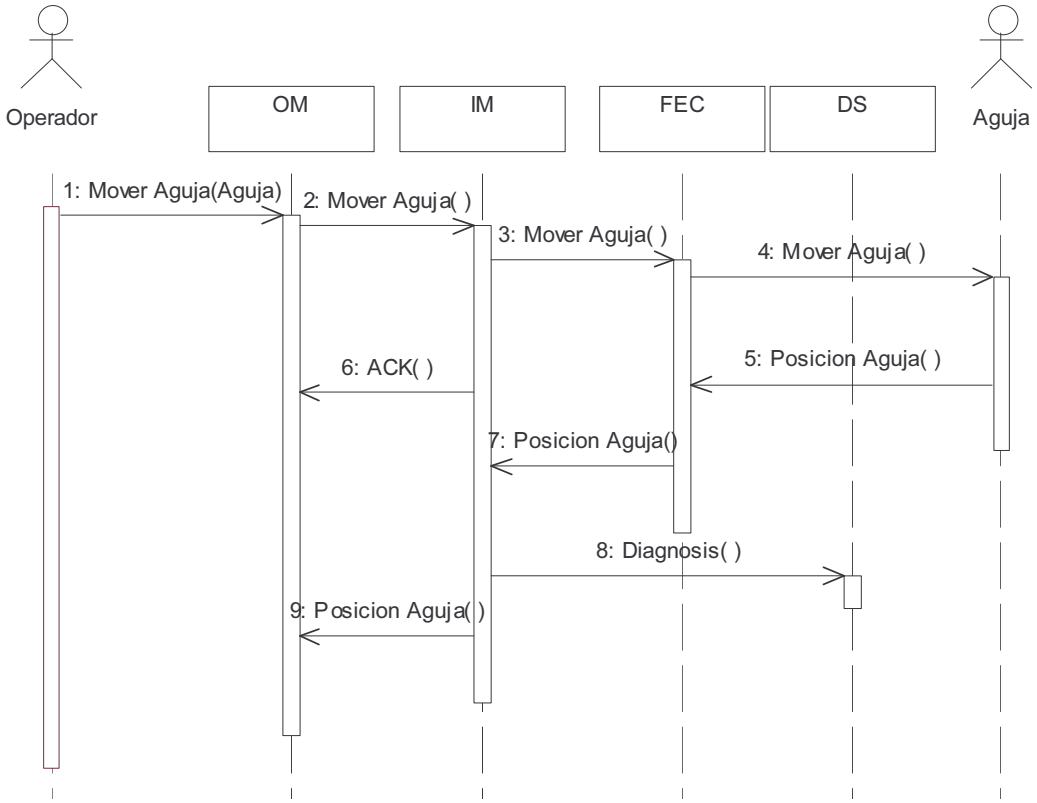


Figura 2.42 Escenario

El escenario o diagrama de interacción asocia a cada objeto una línea vertical. Los mensajes que se intercambian entre los objetos se representan con flechas horizontales desde el objeto emisor al objeto receptor (puede haber varios también). Se puede añadir, pintándolo sobre la línea vertical, el estado

interno de cada objeto, con lo que se consigue relacionar mejor este diagrama con el diagrama de estados propio de cada objeto.

Estos diagramas permiten describir de forma muy intuitiva el comportamiento de varios objetos dentro de un caso de uso, aunque no dan información acerca de la funcionalidad que hay que implementar en cada objeto. Para saber lo que hay dentro de las cajas, necesitamos recurrir a los diagramas de transición de estados.

2.4.7 Pruebas de Escenarios

Los escenarios representan requisitos complejos que involucran a varias entidades del sistema. El problema de los diagramas de secuencia de UML, o los Diagramas de Secuencia de Mensaje, los *Message Sequenced Charts* de la ITU (MSC), es que muestran una sola de las posibilidades de interacción entre los diferentes objetos. Eso conlleva una cierta ambigüedad, y por eso es un modelo insuficiente para realizar casos de prueba. Esto ha sido detectado por Harel, que ha ideado una variante de los MSC, los LSC [DaHa99], donde se añaden además las interacciones explícitamente prohibidas por los requisitos de seguridad del sistema.

Para Harel, en el desarrollo de software debe existir una continuidad total entre la especificación del comportamiento del sistema y la implementación final. Esa continuidad sólo es plenamente garantizada en la medida que los modelos de especificación son directamente ejecutables [Har01], o permiten generar código de forma automática, como se hace en las herramientas STATEMATE y Raphsody, de la compañía iLogix, con la que el propio Harel colabora.

2.4.7.1 Modelo de fallos

La implementación de un escenario puede adolecer de:

- Salidas Incorrectas o inexistentes
- Fallos en los objetos participantes (errores internos, faltan participantes, etc.)
- Errores en el intercambio de mensajes entre objetos (el mensaje no llega, se manda equivocadamente, el mensaje tiene errores, etc.).

2.4.7.2 Estrategias de prueba

[Bin00] propone una solución al problema de la poca aptitud de los escenarios como modelos de prueba construyendo un diagrama de flujo a partir del escenario, lo cual permite formularlo de manera no ambigua. Esta operación ya es capaz por sí sola de revelar ambigüedades en el modelo sin hacer una sola prueba. El diagrama de flujo se construye identificando condiciones, segmentos y ramas. Cuando ya se ha elaborado el diagrama, se divide en ramas independientes y se prueba cada una de ellas al menos una vez, con un caso de prueba. [Bin00] sugiere la forma práctica de hacerlo, marcando cada nueva rama con un color diferente. Si en el diagrama de flujo hay bucles, hay que añadir tres casos más, uno que no entre en el bucle, otro que pase por el bucle al menos una vez, y otro que pase por el bucle el número máximo de veces permitido.

Otra posible alternativa es transformar los escenarios en diagramas de estados, siguiendo los pasos de [Wh00]. [Wh00] entiende que un escenario es una traza de la ejecución del sistema y, por eso, existen varios escenarios que corresponden a las diferentes posibilidades de intercambio de mensajes. Define así tres actividades principales:

- Eliminar las incompatibilidades existentes entre los diferentes escenarios
- Unificar los escenarios en la medida de lo posible
- Simplificar al máximo el diagrama de estados que se haya obtenido como resultado.

[Wh00] añade más contenido semántico a los escenarios añadiendo para cada objeto una serie de variables de estado de ese objeto, que constituyen un vector de estados. Este vector de estados es la entidad que va a manejar el algoritmo de análisis de los diferentes diagramas de estados (inicialmente, hay un diagrama de estados no jerárquico por cada objeto participante en el escenario). El algoritmo elimina los estados duplicados del diagrama examinando el valor de estados para cada nodo. Posteriormente, busca estados que se puedan englobar en un estado padre.

En la misma línea de transformar el escenario en máquina de estados está la contribución de [Alu00], que muestra un método formal de síntesis de máquinas de estados concurrentes a partir de escenarios (*Message Sequence Charts*) que incluye el análisis de los posibles interbloqueos en el escenario.

2.4.8 Técnicas formales de prueba

Existe una abundante producción científica en el campo de las técnicas formales aplicadas a la verificación de modelos de máquinas de estados. Estos aspectos quedan fuera de los objetivos de la Tesis Doctoral, pero se citan algunos ejemplos de aplicación de métodos formales a statecharts que se consideran de interés. El método que se va a proponer asume que el modelo de estados sobre el que se construyen los casos de prueba es formalmente consistente (no existen estados duplicados, ni un mismo evento produce varias transiciones diferentes, etc.).

[Hei96] Presenta el RSML (Requirements State Machine Language) que ha sido utilizado en la especificación de TCAS II, un sistema de control del tráfico aéreo. No utiliza variables internas al sistema para describir su comportamiento, sólo variables externas al procesador (sensores y dispositivos controlados por el software). Los modelos descritos en RSML definen una transición con cinco partes (identificador, origen y destino, máquina de estados donde esta localizada la transición, evento que la dispara, condición de guarda, salida). Los modelos de estados en RSML deciden el siguiente estado mediante una función matemática. Esto tiene la ventaja de evitar el indeterminismo. Los autores exponen una definición formal de las jerarquías de estados de los statecharts y de las transiciones. Esto les permite comprobar que las transiciones definidas no entran en conflicto y que el comportamiento está siempre definido. En la misma línea de trabajo de verificación formal de los modelos ("model checking") está el trabajo de [Day93], aplicado a los diagramas de estados jerárquicos, que ha desarrollado un verificador de máquinas de estados que trabaja conjuntamente con la herramienta de modelado STATEMATE.

Dentro de la prueba de escenarios, [Al00] es un ejemplo de fundamentación teórica de la aplicación de técnicas formales de verificación sobre los MSC (Message Sequence Charts [ITU96]) de la ITU, definiendo las propiedades formales de un MSC y proponiendo un algoritmo que verifica que no se producen situaciones de bloqueo en el escenario. Sería deseable que aparte del algoritmo en notación formal, ofreciera una versión en seudocódigo, pues eso facilitaría la diseminación de sus resultados.

En [Har99] se trata el problema de decidir, dada una especificación LSC (Live Sequence Chart, que es una variante de los escenarios ideada por Harel, que se ha citado anteriormente y en la que se especifican las interacciones que son obligatorias de acuerdo con los requisitos del sistema), si es posible sintetizarla de forma automática. Harel define la consistencia de una especificación LSC como la posibilidad de transformar dicha especificación en una colección de máquinas de estados o statecharts. La técnica que usa es transformar el escenario en una máquina de estados concurrentes (estados AND) para cada objeto participante en el escenario, comprobando que no se producen comportamientos no especificados. Siguiendo la misma línea de pasar directamente de la

especificación al código, en [Bjö00] se da un ejemplo de componentes basados en SDL donde la propia especificación es un modelo fácilmente verificable.

El lenguaje LOTOS es una de las Técnicas de Descripción Formal normalizadas por ISO [ISO89] con una fuerte base matemática basada en Algebras de Procesos que se ha utilizado en la especificación de sistemas de comunicaciones de forma intensiva durante los años noventa. LOLA (LOTOS Laboratory) es una herramienta desarrollada en el DIT de la UPM que soporta diferentes aspectos de LOTOS, concretamente la exploración del espacio de estados y la transformación de las especificaciones LOTOS en máquinas de estados equivalentes [Que89].

2.4.9 UML Testing Profile

Los autores de UML han definido un entorno de arquitectura y componentes de pruebas, el "UML Testing Profile" [OM03] cuyos componentes de prueba (Casos de Prueba, Conjuntos de Pruebas, etc.) se comunican mediante una clase llamada "arbiter" o "árbitro", que es la que gestiona la ejecución de las pruebas. Este modelo se compara en [OM03] con los utilizados por el entorno de libre distribución JUnit y el de la notación de pruebas estándar TTCN [ITU92] (Tree and Tabular Combined Notation). La Figura 2.43 muestra las relaciones entre los diferentes elementos del modelo de UML Testing Profile.

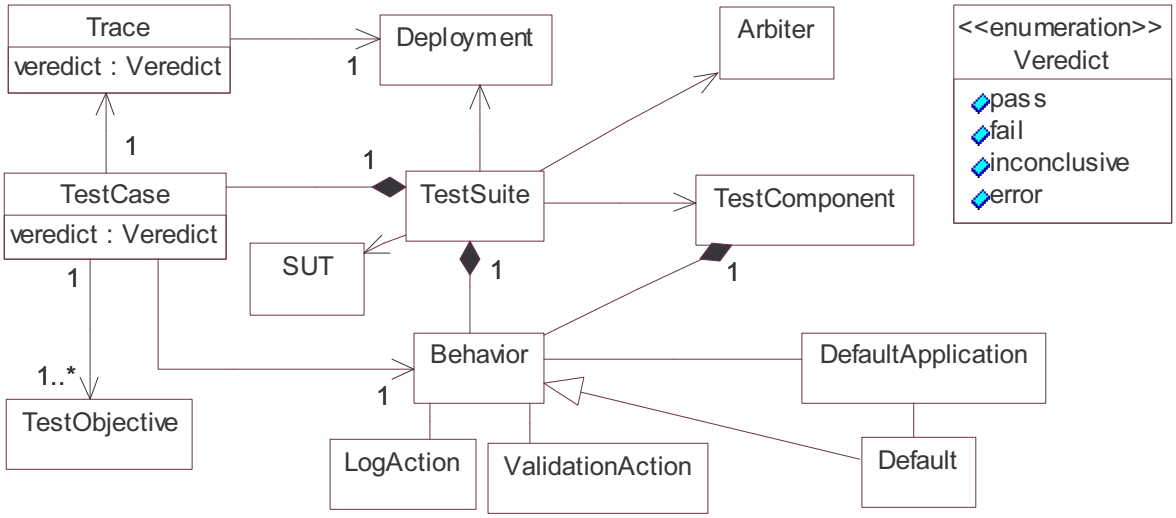


Figura 2.43 Modelo de componentes de prueba en UML Testing Profile

Las ideas más importantes de UML Testing Profile son:

- Separación entre comportamiento de la prueba y la evaluación de la prueba introduciendo el componente "árbitro".
- Integración de los conceptos de control de prueba, grupo de pruebas y caso de prueba en un concepto general de caso de prueba, que a su vez se descompone en casos de prueba de más bajo nivel, lo que permite reutilizar las definiciones de las pruebas en las diferentes jerarquías: un conjunto de pruebas es un caso de prueba del nivel más alto.
- Añadir los "valores por defecto" para gestionar los comportamientos inesperados, los veredictos y las excepciones.

- Soporte de particiones de datos no sólo para las observaciones, sino también para los estímulos: de esta forma se puede describir los casos de prueba de forma lógica sin necesidad de definir los estímulos por completo, sino solo su rango de valores.

Aparte de eso, se ha añadido por motivos prácticos:

- Una configuración de pruebas, como estructura interna de un conjunto de pruebas que se usa para permitir la definición de los componentes de prueba, describiendo su inicialización y sus interfaces.
- Un diagrama de despliegue permite asignar los requisitos respecto de la ejecución de las pruebas en los nodos de la red.

En el entorno "UML Testing Profile" se ofrecen conceptos de

- Comportamiento de la prueba en lo referente a su observación.
- Arquitectura de las pruebas, es decir los elementos y sus relaciones implicados en una prueba.
- Datos de prueba, es decir, la estructura y el significado de los valores procesados en una prueba.
- Tiempo, es decir las restricciones temporales y la medición del tiempo durante la ejecución de las pruebas, definiendo una interfaz para la gestión de los temporizadores.

Sin embargo, por su carácter genérico, UML Testing Profile no ofrece mecanismos específicos para prueba de Líneas de Producto Software, y como está más bien centrado en los aspectos de arquitectura software del entorno de pruebas, tampoco proporciona una pauta para relacionar los casos de prueba con los requisitos.

2.5 Automatización de pruebas

La automatización de las pruebas tiene evidentes ventajas frente a realizar las mismas pruebas manualmente. La automatización merece la pena si las pruebas que van a realizarse van a tener que repetirse un gran número de veces [Bin00]. Ese puede ser el caso de las Líneas de Producto Software, y por ello, hacemos hincapié en estos aspectos. [Few99] cita las siguientes ventajas:

- La más evidente es que se pueden realizar con gran facilidad las mismas pruebas a todas las versiones del software. Esto puede a veces no ser tan útil, si por ejemplo ha habido un cambio de requisitos entre versión y versión, o se han introducido incompatibilidades entre el software de prueba y la aplicación.
- Se pueden realizar más pruebas y con mayor frecuencia.
- Es posible hacer pruebas que no sería posible realizar manualmente. Por ejemplo, pruebas de sobrecarga.
- Se usan mejor los recursos. Por un lado, las personas que hacen las pruebas automáticas no necesitan ser tan expertas como quienes prueban manualmente, y, por otro lado, se descarga de tareas repetitivas y monótonas al equipo de pruebas.
- Las pruebas son más consistentes y repetibles.
- Las pruebas son reutilizables.
- El tiempo de duración de las pruebas es menor.
- Se incrementa la confianza en la calidad del producto final.

Estos beneficios no se logran sin pagar un precio, que puede ser en ocasiones demasiado alto. Posibles problemas son:

- Expectativas no realistas. Automatización no es lo mismo que magia. [Few99] estima que lleva cinco veces más tiempo ejecutar una prueba de forma automática que hacerlo directamente de forma manual.
- Proceso de prueba pobre, mal documentado e inefectivo. En este caso, merece la pena intentar mejorar las pruebas antes que automatizar algo que es ineficiente de por sí.
- Creer que las pruebas automáticas encontrarán muchos fallos. Es dudoso, salvo que se cambie la aplicación. Tiene más sentido como apoyo a las pruebas de regresión.
- Sensación falsa de seguridad. Que un conjunto de pruebas automatizadas pase con éxito, no implica necesariamente que no hay errores en la aplicación. Hay que estar seguros de que las propias pruebas automatizadas no generan resultados erróneos.
- Problemas técnicos asociados con las herramientas de prueba utilizadas.
- Dificultades internas a la organización en la que se quieren implantar las pruebas automáticas.

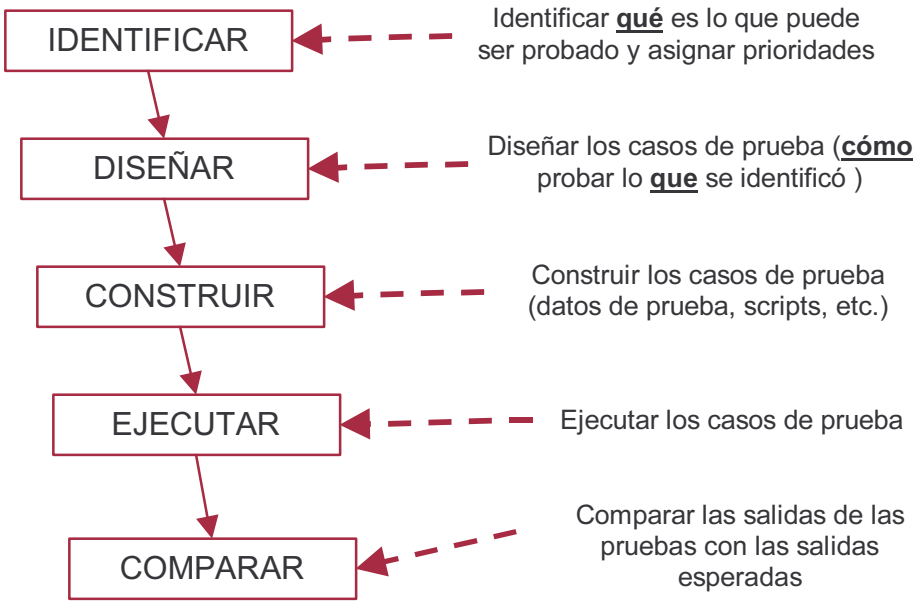


Figura 2.44 Actividades de las pruebas de software

La Figura 2.44 muestra las diferentes tareas que hay en las pruebas del software [Few99]. De estas cinco, las que son más repetitivas y requieren menos esfuerzo intelectual son las dos últimas: ejecutar y comparar. La de construir está a medio camino entre éstas y las dos primeras, que requieren un gran conocimiento del sistema y que no son fácilmente automatizables.

Además, hay que añadir las siguientes actividades:

- Elección de la herramienta de pruebas. Automatizar las pruebas no es implantar sin más una herramienta de moviola o de otro tipo de las que hay en el mercado. [Bin01] sostiene que no hay herramienta de prueba perfecta, y, la elección de una u otra dependerá en cada caso de las características de la aplicación y de la organización en la que se vaya a implantar.
- Diseño de arquitectura del "testware" o entorno de prueba automática, incluyendo una estructura de directorios para almacenar las distintas herramientas de soporte, los ficheros asociados con los casos de prueba, y los resultados de las ejecuciones de las pruebas.
- Generación de los "scripts" o guiones de prueba. [Few99] indica, que según su experiencia, hay que buscar antes que nada la mantenibilidad y la flexibilidad. Deben ser fácilmente legibles,

incorporando comentarios y con el mayor nivel de abstracción posible. Cuanto más independientes sean de la implementación, más gente podrá entenderlos y modificarlos.
- Ejecución de las pruebas. Los resultados se pueden evaluar en tiempo de ejecución o posteriormente con la ayuda de una herramienta.
- Mantenibilidad de las pruebas. Hay que preocuparse de que los casos de prueba no sean redundantes ni queden obsoletos. Deben de evolucionar paralelamente a la aplicación.

2.5.1 Herramientas Software de prueba automática

Actualmente existen en el mercado múltiples productos que ayudan en la tarea de automatización de las pruebas. No se pretende ahora enumerar todas ellas exhaustivamente ni hacer una comparativa entre las más importantes. La herramienta universal no existe [Bin00]. En primer lugar, porque al automatizar pruebas lo primero que hay que plantearse es cómo son las pruebas que se quiere automatizar y cómo se quiere hacer, y, en función de esto elegir la herramienta. Hacer lo contrario, es decir, comprar una herramienta y luego empezar a pensar cómo automatizar es comenzar a construir la casa por el tejado [Few99].

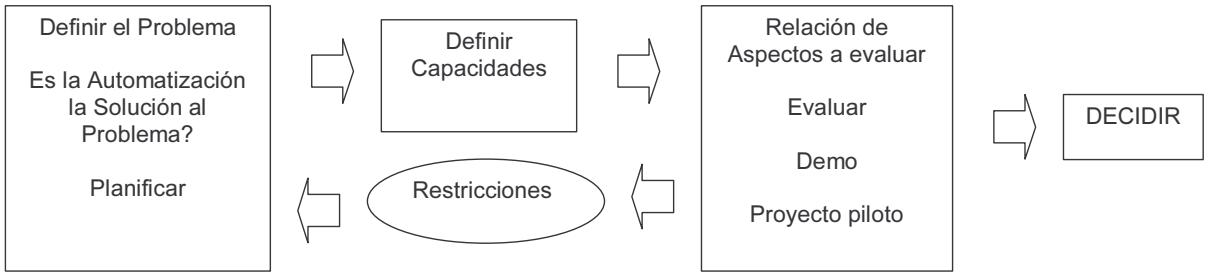


Figura 2.45 Esquema del proceso de selección de una herramienta de pruebas

Una vez que se tiene claro qué pruebas se va a automatizar, puede comenzar el proceso de selección de una herramienta de pruebas. [Few99] recomienda, siguiendo el esquema de la Figura 2.45 identificar las restricciones que condicionan la elección de la herramienta centrando el análisis en los siguientes puntos:

- Entorno de aplicación de la herramienta hardware y software.
- Plataforma donde debe de ejecutarse la herramienta.
- Soporte comercial.
- Coste.
- Factores políticos (obligatoriedad de comprar a determinados fabricantes, etc.).
- Calidad de la herramienta (documentación, complejidad, frecuencia de fallos, riesgo de que la herramienta provoque fallos en otras partes del entorno, etc.).

La compañía india RelQ [Sat02], que está especializada en validación y verificación de sistemas software de aviación, trabaja con un proceso algo diferente, y que se recoge en la Figura 2.46.

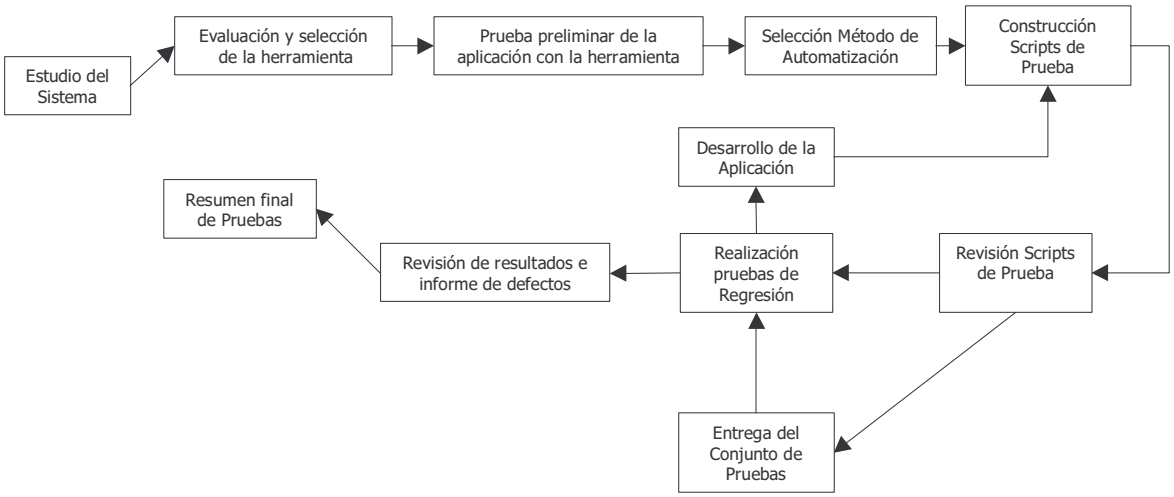


Figura 2.46 Modelo de automatización de Pruebas

Si no se encuentra ninguna herramienta que satisfaga los requisitos deseados, se puede plantear la decisión de desarrollar la herramienta de forma interna en lugar de comprarla. Es evidente que no se puede pretender igualar la complejidad de una herramienta comercial con un desarrollo interno, pero si se delimita adecuadamente el ámbito de aplicación de la herramienta y se dispone de los recursos necesarios, puede lograrse un resultado óptimo.

Una vez que se ha elegido una herramienta, ya sea siguiendo el esquema de la Figura 2.45 o de otra forma, comienza la implantación de la herramienta de pruebas dentro de la organización. Es una tarea ardua, donde los problemas técnicos suelen ser menores frente a los de carácter organizativo. [Few99] dice que se necesitan dos cosas para lograr el éxito: que la dirección apoye la implantación y que haya al menos una persona que se involucre activamente.

Para terminar este subcapítulo, se ofrece una breve relación de herramientas para pruebas de software presentes en el mercado y el soporte que dan a la prueba de Líneas de Producto. Esta tabla no busca ser una evaluación detallada de las herramientas citadas. Este tipo de estudios, que muchas veces son proyectos de consultoría cuyos resultados no son públicos, se queda obsoleto con rapidez por la propia dinámica del mercado y, como se ha dicho ya, las herramientas son buenas o malas según para lo que se las quiera utilizar.

Herramienta	**Fabricante**	**Soporte de L. P. posible**	**Aplicación principal**	**Esfuerzo estimado**
Attol	Rational	SI	Pruebas unitarias	Medio/Alto
Cantata	IPL	SI	Pruebas unitarias	Medio/Alto Configuración complicada
Jameleon	Dominio Público	SI	Entorno integrado de pruebas. Test cases especificados de forma abstracta mediante XML	Medio/Alto
JUnit	Dominio Público	SI	Pruebas unitarias	Alto

Herramienta	Fabricante	Soporte de L. P. posible	Aplicación principal	Esfuerzo estimado
Rational Suite	Rational	SI	Entorno integrado de pruebas. Incluye varias herramientas con diferentes propósitos	Medio/Alto Asocia Requisitos a Casos de Prueba
INQ	Integri	SI (en su sector)	Pruebas automáticas de terminales para tarjetas de pago electrónico	Medio/alto
Ruby	Dominio Público	SI	Lenguaje de script para formulación de casos de prueba de gran potencia	Alto
Telelogic Tau	Telelogic	SI	Pruebas a partir del modelo del sistema en UML, usando el lenguaje formal SDL y la notación de pruebas TTCN (Tree and tabular Combined Notation)	Medio/Alto Necesario conocer TTCN
Win Runner	Mercury	SI	Prueba de aplicaciones Windows. Reutilización de pruebas soportada	Medio/Alto
Test Director	Mercury	SI	Entorno de gestión de pruebas: Gestión de Requisitos Pruebas gestión de defectos Se usa en combinación con WinRunner	Medio/Alto
Test Work Flow	SQS	SI	Validación de sistemas de tiempo real, relacionando casos de prueba con requisitos	Medio/Alto Herramienta pensada para validar sistemas de tiempo real
Testscope	Scope GmbH	SI	Análisis Estático Pruebas Unitarias Cobertura de código	Medio/Alto

Tabla 2.10 Herramientas Comerciales de Prueba Software

La selección de herramientas se ha hecho partiendo de la información que proporcionaban los distintos vendedores de herramientas software presentes en la 3rd ICSTEST International Conference on Software Testing (Düsseldorf, Alemania, Abril de 2002) y se ha actualizado posteriormente. Incluye grandes fabricantes de herramientas complejas y poderosas como Rational (ver en la tabla siguiente la relación completa de herramientas de este fabricante), Mercury y Telelogic, y algunas otras a las que se ha tenido acceso por diferentes motivos, entre las cuales habría que destacar la herramienta Test Work Flow, orientada a sistemas de tiempo real, desarrollada en España por la consultora SQS, por implementar un proceso de pruebas que integra la gestión de requisitos, el diseño de las pruebas y la ejecución.

<table>
<tr><th colspan="3">RATIONAL SUITE</th></tr>
<tr><td>Requisitos y Análisis

• Desarrollo de requisitos y Casos de Uso: Requisite Pro
• Modelo de Negocio: Rose
• Modelado de Datos: Rose</td><td>Desarrollo Software

• Modelado Visual:Rose
• Pruebas Unitarias:
Purify
QualityArchitect
Test Real Time
• Entorno de desarrollo: XDE</td><td>Pruebas de Sistema

• Funcionalidad: Robot
• Fiabilidad:
Purify
PureCoverage
• Tiempo Real: Test Real Time</td></tr>
<tr><td colspan="3">Gestión Integrada de Proyectos

• Proceso Unificado RUP
• Gestión de Documentación: SoDA
• Métricas de Proyecto: ProjectConsole</td></tr>
<tr><td colspan="3">Trabajo en Equipo

• Gestión de Requisitos: RequisitePro
• Gestión de Pruebas: TestManager
• Gestión de Configuración: ClearCase, ClearQuest
• Gestión de Contenidos: ContentStudio</td></tr>
</table>

Tabla 2.11 Entorno de Herramientas Rational Suite

En una primera aproximación, para soportar pruebas automáticas de Líneas de Producto, lo que se necesita es una herramienta que permita clasificar los casos de prueba para decir cuales de ellos se ejecutan sobre un producto determinado, cuáles no y cuáles son comunes. En este sentido, son igualmente válidas todas las herramientas que se citan a continuación. Si la herramienta ofrece ya por sí misma una cierta capacidad de clasificación de las pruebas, deberá intentar utilizarse esta posibilidad y adaptarla a las propias necesidades, y si no existe, habrá que implementarla de otra forma (con una estructura de directorios en la que se organicen los artefactos de prueba, por ejemplo). Es esencial que la herramienta sea adecuada a las características del software que pretende probarse con ella. Difícilmente se va a poder utilizar una herramienta de pruebas de aplicaciones gráficas para probar aplicaciones de software empotrado.

Como se ve en la relación de herramientas previa, no existe ninguna herramienta que, a priori, sea incompatible con las Líneas de Producto. Todas estas herramientas permiten definir un conjunto de entidades software (ficheros, etc.) sobre las que se van a hacer las pruebas. La línea de productos software, al igual que cualquier sistema software, ya tiene de por sí una organización de sus componentes. Hablando en términos muy simples, habrá un código fuente que implemente la parte común y otro código que implemente las diferentes variaciones existentes. La organización de ese código en ficheros, librerías, etc. puede adoptar múltiples formas. Las Líneas de Producto Software no están vinculadas necesariamente a un lenguaje de programación determinado. Como posibilidad, podría existir una línea de productos en lenguaje ensamblador. Evidentemente, en este caso, no sería posible usar mecanismos como la herencia, las clases abstractas, etc. que están disponibles en los lenguajes orientados a objetos y que permiten modelar fácilmente la variabilidad de las Líneas de Producto. Y siguiendo con el ejemplo, tampoco habría muchas herramientas de pruebas en el mercado disponibles, pero no por tener una Línea de Producto Software, sino por el lenguaje de programación utilizado.

La integración de una herramienta de pruebas dentro del desarrollo de una línea de productos es algo necesariamente específico de cada proyecto software, pues depende necesariamente de la arquitectura del sistema y de su implementación. Por lo tanto, a la hora de planificar un proyecto de desarrollo software, hay que prever un paquete de trabajo de implantación de la herramienta de pruebas.

2.6 Prueba de Líneas de Producto Software

Las pruebas de una línea de productos, deben según [ClNo01] examinar los activos básicos, el software específico de los productos, y las interacciones entre ellos. Los responsables de las pruebas pueden ser diferentes para cada uno de ellos dentro de una organización, como en [MeDu01]. [GrSy00] comenta que, aunque lo habitual es que las pruebas de sistema y de integración estén bajo la responsabilidad de quien desarrolla los productos, el equipo responsable de los activos básicos debe de realizar unas pruebas unitarias tan exhaustivas como sea posible. Es necesario también establecer unos mecanismos de gestión de defectos, para resolver los posibles problemas de integración con la plataforma que pudieran existir.

[ClNo01] da algunas recomendaciones básicas para las pruebas de las Líneas de Producto, aunque las tres últimas son válidas para cualquier prueba software en general también:

- Estructurar el software de prueba de forma que sea reutilizable. La trazabilidad entre casos de prueba y arquitectura, es según [ClNo01] un medio para alcanzar esto, ya que al cambiar una parte del código, no hay más que cambiar sus casos de prueba asociados. Eso es posible siempre que la estructura de la aplicación lo permita. Si se aísla la parte de software pruebas de servicios generales (grabación de resultados, cálculos auxiliares, etc.) se puede volver a emplear en nuevos casos de prueba muy fácilmente.
- Incluir en la propia arquitectura servicios que faciliten las pruebas. Por ejemplo, acceso a determinadas variables de estado, rutinas de autocomprobación, etc.
- Basar las pruebas en los requisitos y en la definición de la arquitectura.
- Utilizar pruebas de regresión.
- Usar un entorno de pruebas lo más parecido al entorno final del sistema.

Los artefactos empleados en las pruebas (documentos, conjuntos de datos de prueba, herramientas software) de los activos básicos de la línea de productos pueden reutilizarse también en toda la línea de productos. Lo mismo puede decirse para las pruebas de los productos.

No se puede afirmar de forma general que las pruebas puedan reutilizarse en una línea de productos. Dependerá de las características del dominio, de los recursos disponibles, y de la arquitectura de la línea de productos. Si podemos aislar partes en la arquitectura, probarlas por separado y luego ensamblarlas sin temor a efectos laterales, será viable omitir algunas de las pruebas en determinados productos. Si no es así, habrá que repetir todas las pruebas. En este tipo de desarrollos, se requiere un gran esfuerzo en las pruebas unitarias y de integración. El esfuerzo en pruebas de regresión es en principio, importante.

[Wei01] concibe el proceso de desarrollo de una línea de productos partiendo de un modelo de desarrollo en V convencional para un único producto (Figura 2.47). Las flechas discontinuas representan la trazabilidad: tanto de los requisitos, casos de uso y escenarios en los elementos de la arquitectura y del diseño (trazabilidad vertical) como de los requisitos, casos de uso y escenarios en los casos de prueba (trazabilidad horizontal).

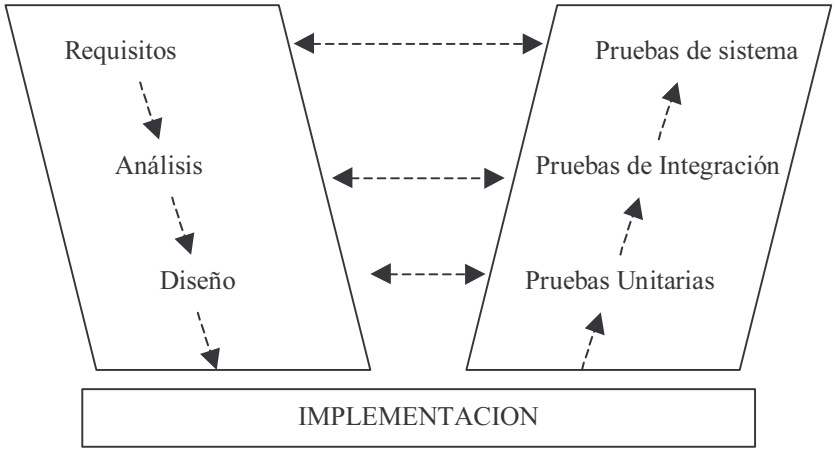


Figura 2.47 Modelo de ciclo de vida en V

En las familias de productos, según [Wei01], hay que tener además en cuenta que hay una fase añadida de ingeniería del dominio, para definir aspectos comunes y específicos; se debe de diseñar para poder reutilizar; los cambios son más frecuentes y complejos; y se debe de tener en cuenta la arquitectura y componentes existentes (si los hubiera). Estos factores determinan un nuevo modelo del ciclo de vida que se representa en la Figura 3.1. [Wei01] añade al ciclo de vida convencional la fase de ingeniería del dominio, haciendo hincapié en la importancia de la trazabilidad de todos los elementos del proceso, especialmente de los casos de prueba, respecto a los elementos del análisis del dominio (casos de uso y escenarios). [Wei01] señala también que, en realidad, existen dos ciclos de vida: el desarrollo de la familia y el desarrollo de los productos.

[Wei01] dice que la ingeniería de familias de productos sirve para obtener software de mayor calidad al reutilizar componentes que se han probado previamente, pero sólo si el impacto de los cambios se ha analizado con detalle y rigor.

[Neb02] presenta una metodología de prueba para Línea de Productos Software utilizando escenarios expresados en UML, que es interesante como proceso porque integra la especificación de requisitos con la especificación de casos de prueba. Se define por cada Caso de Uso un escenario describiendo el comportamiento que se quiere probar, y opcionalmente, se define un escenario de inicialización ([Neb02] lo llama *Prefix*) y uno o varios escenarios de "Rechazo", en los que se describe que interacciones no se quiere probar porque no son relevantes para el caso de prueba. [Neb02] propone especificar el caso de prueba con un grado de abstracción suficiente que permita aplicarlo a los diferentes miembros de la Línea de Productos Software.

En [ReMo03] se relata una experiencia industrial de gran interés en la validación de una Línea de Producto de equipos de control de tren por parte de la empresa Bombardier según el estándar CENELEC [CEN97], lo que significa seguir el modelo de ciclo de vida en V. Estos equipos constan de una parte genérica de hardware y software, que necesita ser adaptada para cada tipo de locomotora y que debe de probarse y certificarse por separado. Habitualmente constan de varios subsistemas que necesitan su certificación de seguridad y su validación como subsistema independiente. A fin de aumentar la eficiencia del proceso, se estructura la validación y certificación de la seguridad en dos fases (se hacen varias "V"): producto genérico y producto final específico. De cara a la certificación de seguridad, para derivar de la parte genérica cada producto específico los desarrolladores deben de seguir unas reglas que garantizan la seguridad del producto final. Sin embargo, en cuanto a la validación [ReMo03] no aporta detalles sobre el grado de aplicabilidad de las pruebas funcionales del producto genérico a cada producto específico. Para estructurar los casos de prueba, [ReMo03] utiliza

la técnica de los árboles de clasificación, si bien en el artículo no detalla cómo se gestionan con dicha técnica las variaciones en la Línea de Producto.

La pregunta que se planteaba al principio de este apartado, sobre si el desarrollo de software en líneas de producto es ventajoso a la hora de las pruebas no tiene una respuesta generalizable a todas las situaciones posibles. Puede afirmarse sin embargo que:

- Las pruebas que se hagan de los activos básicos de la línea de productos pueden reutilizarse, es decir hacerse sólo una vez para un producto y darse por superadas para los demás, sólo en función de la arquitectura del sistema. Si los componentes tienen interfaces sencillas y definidos y sus datos internos están aislados, será más fácil garantizar que cambiando algún componente para obtener un nuevo producto, esto no influye en la funcionalidad global del sistema. A veces, en determinados dominios, esto no es posible. Por ejemplo, en los sistemas con requisitos críticos para la seguridad, tanto los suministradores como las autoridades de certificación correspondientes son muy reacios a aceptar esto y prefieren repetir las pruebas para cada nueva entrega de software.

- Si los activos básicos de la línea de productos son módulos con encapsulación de datos (los demás componentes no acceden a sus datos internos directamente, sino sólo a través de las funciones de acceso que les da el componente), y sus interfaces con el resto de la arquitectura están definidas, las pruebas unitarias de cada uno de estos componentes van a poderse reutilizar, puesto que apenas van a variar de un producto a otro. De ahí la conveniencia de concentrar dentro de la arquitectura lo más posible los aspectos de variabilidad de la línea de productos.

- A mayor grado de exhaustividad de las pruebas unitarias de los componentes, menor riesgo de que fallen al integrarse en la arquitectura final. Siempre existirá el riesgo de fallos en la especificación de las interfaces y de que haya problemas de rendimiento al integrar el hardware y el software en el sistema completo.

- Siempre se va a poder reutilizar al menos los casos de prueba y su proceso de generación. En la medida en que la diferencia entre los productos sea menor, la reutilización tendrá mayor alcance.

- La trazabilidad entre requisitos y casos de prueba va a ser en principio, ventajosa. Si es posible identificar los requisitos comunes a toda la familia de productos, quizá sea posible diseñar casos de prueba generales que estén asociados a esos requisitos genéricos. O también, obtenemos de forma inmediata los casos de prueba para un nuevo producto a partir de sus requisitos. Por otro lado, conseguir trazabilidad implica un esfuerzo que debe de evaluarse para ver si es suficientemente rentable.

2.6.1.1 Prueba de los componentes reutilizables

Según [Bin00], reutilizar es una solución general a problemas habituales en el desarrollo de software: elevado coste del desarrollo, fallos inaceptables, baja mantenibilidad y poca flexibilidad. El desarrollo de software reutilizable entraña dificultades adicionales, pues hay que tener en cuenta todas las posibles variaciones.

La aportación más importante y obvia que las pruebas hacen a la reutilización del software es que detectan los fallos en los componentes que se quieren volver a usar. De forma indirecta, cuando en el desarrollo se tiene en cuenta el diseño de las pruebas y el facilitarlas, el diseño se mejora.

En el desarrollo de componentes para Líneas de Producto serían relevantes las siguientes posibilidades:

- Prueba de una clase (o componente) abstracta: Los componentes finales se derivan del componente genérico mediante herencia. Se prueba instanciando el componente abstracto de forma que se cubran en la medida que se estime razonable las posibilidades que ofrece la interfaz abstracta del componente o clase. Se deben de buscar fallos de diseño como identificación de responsabilidades del componente inconsistente o incompleta; interfaz del componente inconsistente con sus responsabilidades, funcionalidades requeridas que no se han incluido, etc.

- Prueba de una clase (o componente) genérica: El componente es configurable por los usuarios asignando valores a determinados parámetros. Los fallos que pueden presentarse estarán ligados a los parámetros de configuración (rangos de los valores, diferentes tipos de datos admitidos, robustez frente a tipos de datos erróneos o no contemplados, etc.). Este tipo de pruebas puede automatizarse implementando un componente que vaya ensayando diferentes posibilidades.

- Prueba de una nueva plataforma: Un conjunto de activos básicos se desarrolla de nuevo y debe de ser probado. Los fallos que deben de buscarse van desde omisiones en el diseño, inconsistencia con el modelo del dominio, estructura confusa y de interacción con los componentes específicos de cada producto. Todos los pasos que se den tendientes a automatizar las pruebas, van a poder volverse a emplear en futuros desarrollos.

- Prueba de una plataforma ya empleada: Existe ya una base de productos que funcionan con una versión anterior de los activos básicos y hay que comprobar que la nueva versión se integra con los productos existentes de forma adecuada. Los posibles errores son, a saber, problemas de incompatibilidad entre componentes nuevos y antiguos, fallos latentes que salgan a la luz con la nueva versión, y efectos laterales variados entre funcionalidades nuevas y antiguas. Un terreno, en definitiva, donde la realidad supera con creces la ficción. En este tipo de pruebas, un entorno automatizado es de gran ayuda para reducir el esfuerzo manteniendo la fiabilidad de las pruebas.

3 Método de Pruebas de Líneas de Producto Software de Tiempo Real

3.1 Objetivos y Alcance

El objetivo principal del método es proporcionar una pauta de trabajo para diseñar, implementar y ejecutar casos de prueba sobre una Línea de Producto Software partiendo de su Especificación de Requisitos.

El método tiene como trasfondo la experiencia práctica del autor de la Tesis Doctoral en las Pruebas de Validación de sistemas ferroviarios de acuerdo a la norma CENELEC [CEN97] [Br03], que se detalla en la parte de aplicación práctica de la Tesis Doctoral, donde hay que demostrar que las pruebas cubren todos los requisitos, especialmente los que afectan a la seguridad.

Se desea verificar la conformidad de una implementación respecto a su Especificación de Requisitos, con la particularidad de que la implementación es una Línea de Producto Software y en vez de una implementación se esta hablando en realidad de varias implementaciones similares.

Además, la implementación bajo prueba o IUT (*Implementation under Testing*) es un Software de Tiempo Real, lo que implica que parte de los requisitos van a ser tiempos de respuesta, etc. La literatura sobre estos sistemas [Dou99] los clasifica en:

- Sistemas de tiempo real “duro” o estricto (*hard realtime*): Para una determinada acción del sistema hay tiempo límite que si se rebasa es un error.
- Sistemas de tiempo real ”blando” o flexible (*soft realtime*): No se tiene que rebasar en media el tiempo límite.
- Sistemas de tiempo real “firme” (*firm real time*): No se tiene que rebasar en media el tiempo límite, pero hay un valor de desviación máximo.

Se pretende que el método sea aplicable a cuantos más sistemas de tiempo real mejor, pero el caso concreto de aplicación que se ha incluido en la Tesis Doctoral es el de un sistema de tiempo real “firme”. La aplicabilidad del método a sistemas de tiempo real estricto dependerá de si en el sistema que se aplique el método es posible monitorizar el comportamiento del sistema sin alterar su comportamiento a nivel temporal de forma significativa.

El método de pruebas busca verificar sobre todo los requisitos funcionales de la implementación bajo prueba y la contempla como una caja negra, por ello los aspectos de los sistemas de tiempo real que afectan a la estructura interna del software como concurrencia de procesos y hebras, planificación de tareas y gestión de memoria no son objeto directo del método de pruebas.

3.1.1 El método dentro del Modelo de Ciclo de Vida en V extendido para Líneas de Producto Software

Se ha estudiado el proceso de desarrollo de Líneas de Producto Software en general y a continuación se ha abordado más en concreto lo referente a las Líneas de Producto Software en la fase de pruebas. Para ello se han recogido las aportaciones que se han considerado más significativas en la literatura.

El método de pruebas de software que se va a presentar concibe las pruebas como un proceso paralelo al desarrollo y fuertemente interrelacionado con él. En este sentido, sigue la orientación propuesta entre otros por [Bin00], [ClNo01] y [GrSy01], además de las experiencias de carácter industrial relatadas en la prueba de Líneas de Producto Software en [CA03].

Una de estas experiencias industriales de la compañía Philips recogida en [CA03] consiste en el desglose del modelo en V de ciclo de vida del software tradicional en varias V independientes para cada uno de los "activos básicos" o componentes de la Línea de Productos Software donde la especificación, el desarrollo y las pruebas de cada "activo básico" tienen lugar de forma simultánea. Se define adicionalmente una fase de integración de los diferentes componentes y una interfaz estándar que hace posible las pruebas de los "activos básicos" de forma automática. [CA03] también destaca la importancia de usar en las pruebas métodos comunes para toda la Línea de Producto Software, además de infraestructuras de pruebas (laboratorios, entornos) y plataformas hardware lo más uniformes y estándares posible.

[CA03] recoge también otra experiencia interesante en Prueba de Líneas de Producto Software de la compañía Siemens y la Universidad de Essen definiendo a partir de los requisitos un conjunto de escenarios de los cuales se derivan los casos de prueba y proponiendo un modelo en V extendido similar al de [Wei01]. Esto impone una estructuración de los Casos de Uso de tal forma que sean fácilmente reconocibles las precondiciones, las poscondiciones, el flujo de ejecución normal, los flujos de ejecución alternativos, si existen, y las excepciones. Los pasos de los Casos de Uso deben de ser atómicos, de forma que un paso se pueda asociar unívocamente con un flujo de ejecución.

Volviendo al modelo del ciclo de vida en V clásico al que se ha hecho referencia al principio de la Tesis Doctoral, se puede decir que el objetivo básico del método de pruebas que se propone sería garantizar la trazabilidad de requisitos y casos de prueba, realizando lo más paralelamente posible la definición del sistema con la especificación de las pruebas. Como se verá más adelante con mayor detalle, de la misma forma que van a existir requisitos genéricos y específicos, se va a replicar la misma estructura en los Casos de Prueba derivados de los Requisitos.

[Sch02] describe cuatro modelos de proceso de desarrollo posibles de Línea de Producto Software:

- Independiente: Se planifica partiendo de cero un conjunto de productos y su arquitectura común (lo que es raro y a la vez arriesgado, pues se puede invertir tiempo en características que luego no hagan falta).
- Integración de Proyectos: Se unifican incrementalmente en una Línea de Producto Software varios proyectos semejantes y que transcurren simultáneamente en el tiempo.
- Reingeniería de sistemas: Se unifican varios sistemas existentes, lo cual aparte de que puede ser no trivial, se justifica sólo si la Línea de Producto Software va a permitir reducir el esfuerzo en desarrollos futuros, porque no se va a obtener un producto mejor con la Línea de Producto Software, sino algo que hace lo mismo que hacía el sistema antiguo.
- Expansión a nuevos ámbitos: Se añaden nuevas características para cubrir un nuevo campo de aplicación utilizando una Línea de Producto Software ya existente, que es el caso más complejo pues es donde hay que tener en cuenta más factores.

Se va a asumir entre todos los posibles, el de Integración de Proyectos como el que se puede presentar en la práctica de forma más habitual. Es decir, existe un núcleo o "kernel" genérico, formado por los activos básicos o parte genérica de la Línea de Producto Software, que incorpora unas determinadas características o *features*, y que se va ampliando con nuevas características conforme van surgiendo las diferentes variantes.

El modelo de ciclo de vida resultante aparece en la Figura 3.1 y matiza el propuesto por [Wei01].Por tratarse de una Línea de Producto Software, existe una fase adicional de ingeniería del dominio, en la que se consideran en conjunto todas las posibles variaciones, definiendo los escenarios y casos de uso que son aplicables a todos los productos y la lista de requisitos globales o características, que es un esbozo de las diferencias de cada producto respecto del producto genérico formado por los llamados *core assets* o activos básicos de la Línea de Producto Software.

En los Casos de Prueba Derivados del Diseño, que son las pruebas unitarias, se puede diferenciar Pruebas Unitarias Genéricas, y Pruebas Unitarias Específicas, según se esté probando un componente software que sea un "activo básico" de la Línea de Productos Software o no. En cualquier caso las pruebas unitarias de un componente software son iguales si el componente pertenece a una línea de productos software o a un sistema software que se desarrolla de forma individual. Las pruebas unitarias de Líneas de Producto Software tienen la peculiaridad en el caso de los "activos básicos" de que deben de ser válidas para toda la Línea de Producto Software, lo cual es posible sólo si los datos internos de los "activos básicos" están encapsulados y sus interfaces son sencillas y definidas.

Las pruebas de sistema y las pruebas de integración cambian respecto al modelo de ciclo de vida en V tradicional para poder gestionar la variabilidad de la Línea de Productos Software. Si existen variantes en los requisitos, habrá diferentes pruebas de sistema para cada variante, y lo mismo con las interfaces y las pruebas de integración. Se define además una nueva actividad dentro de las pruebas de sistema para verificar el mecanismo de obtención de las diferentes variantes.

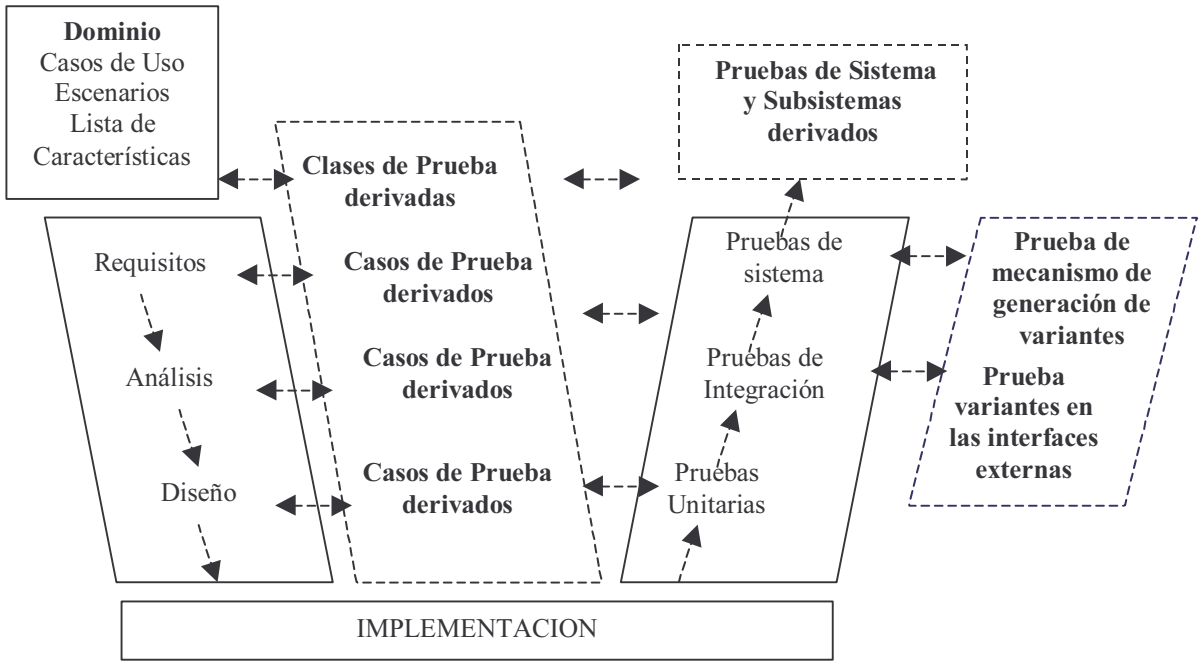


Figura 3.1 Modelo de ciclo de Vida en V extendido

La base de las pruebas es siempre la especificación de requisitos, y los demás elementos como Arquitectura de Sistema y Diseño se pueden utilizar como información para formular los casos de prueba, pero no de forma obligatoria: se asume que la implementación debe de ajustarse a la especificación de Requisitos y se prueba que la implementación cumple dicha especificación. Lo contrario (probar la implementación contra sí misma) es una práctica desaconsejada y poco efectiva para que los fallos salgan a la luz [Pos98].

En el análisis de dominio se identifican las entidades u objetos del sistema que se van a modelar mediante diagramas de estados y que van a servir para definir las *Clases de prueba*, donde se van a agrupar los casos de prueba y sus requisitos asociados.

Esta forma de proceder tiene las ventajas de que se puede especificar las pruebas a la vez que se diseña e implementa el sistema y que la formulación de los Casos de Prueba derivados puede detectar fallos en la especificación de requisitos, en la arquitectura y en la implementación. En el caso de la especificación de requisitos, los Casos de Prueba derivados pueden sacar a la luz requisitos incompletos o inconsistentes.

Sin embargo, no todo son ventajas, pues implantar un ciclo de vida software con consistencia entre requisitos y casos de prueba supone un esfuerzo importante, tanto por lo que es el propio trabajo en sí como por el esfuerzo de incorporar este procedimiento dentro de una organización con unas pautas de funcionamiento ya consolidadas y no necesariamente propicias. La alternativa, llevada al extremo, es que las pruebas se basen en la experiencia de quienes las realizan y en requisitos no documentados, que implica un mayor riesgo de que los errores se descubran con el sistema en explotación y sea más costoso corregirlos.

3.1.2 Objetivo principal del método: las pruebas de sistema

La parte de las pruebas que se está pretendiendo cubrir con el método son las pruebas de sistema incluyendo dentro de las mismas las pruebas de integración de las interfaces externas. Dichas pruebas se derivan de la Especificación de Requisitos del Sistema y no tienen en cuenta su estructura interna, sino que lo “ven” como una “caja negra”.

La relación con los requisitos no es tan clara en el caso de las pruebas unitarias, y por eso se dejan dichas pruebas fuera del alcance del método. Las pruebas unitarias tienen una mayor dependencia de la implementación, y, aunque se haya conseguido tener trazabilidad entre el código y los requisitos, en las pruebas unitarias no se tiene la perspectiva de sistema completo. Por ejemplo, si se está probando de forma unitaria una función que calcula un código de redundancia, y el requisito de sistema asociado a esa función es que los errores en determinada interfaz de comunicación han de detectarse; no se puede decir a priori que si la función pasa la prueba unitaria, el requisito de sistema se vaya a satisfacer.

3.1.3 Justificación de la inclusión de la automatización en el Método de Pruebas

Se asume un contexto restrictivo en lo referente a posibilidades de reutilización de las pruebas y con requisitos de un nivel exhaustivo en cuanto a las pruebas de regresión. Esto significa que las pruebas no se puedan obviar en un producto nuevo pese a que se hayan realizado sobre otro de los componentes de la línea de productos. Esto ocurre en algunos dominios de aplicación con reglamentaciones particulares para la comprobación de la seguridad del software, en los que el cliente o el organismo certificador correspondiente no considera aceptable dejar de realizar determinadas pruebas, aunque la organización suministradora del producto software asuma toda la responsabilidad.

Lo único reutilizable va a ser el proceso de prueba y los casos de prueba, pero las pruebas van a tener que ejecutarse todas para cada producto. Para poder optimizar el esfuerzo en esta situación, es necesario conseguir un cierto grado de automatización en las pruebas.

3.2 Conceptos Generales del Método de Pruebas

Para dar un mayor rigor al método de pruebas se utiliza el concepto de conformidad presentado en [ITU95],[ITU97] y [Hue95]. Junto al concepto de conformidad se introducen además el de analogía, como relación existente entre dos miembros de una Línea de Producto Software, y el de Trazabilidad, por su papel clave en el método para definir el proceso de prueba y estructurar los casos de prueba.

3.2.1 Conformidad

Conformidad significa que una implementación se ajusta a su especificación. Para precisar el concepto de conformidad, se elaboran modelos formales de las implementaciones y así la conformidad puede definirse mediante relaciones entre los modelos de las implementaciones y la especificación, por la satisfacción de requisitos por parte de los modelos de la implementación, o por ambas cosas a la vez.

La noción convencional de conformidad que se usa, por ejemplo en telecomunicaciones, de conformidad de una implementación con un protocolo determinado no es la empleada por el método propuesto en la Tesis Doctoral. Se asume que la especificación no es un estándar único sino que puede ser un conjunto de requisitos definido a partir de por ejemplo, una norma técnica de carácter general, documentos diversos del cliente, etc. En el trasfondo de esta decisión están la experiencia profesional del autor de la Tesis Doctoral y el ejemplo escogido para validar el método propuesto, que es un sistema de control ferroviario, cuyos requisitos se definen a medida del proyecto concreto entre el cliente y la empresa adjudicataria basándose en los reglamentos de circulación ferroviarios generales y otros estándares relativos a calidad, seguridad, etc.

Una especificación (formal) prescribe el comportamiento de un sistema usando una técnica de descripción formal (FDT). Si la técnica de descripción formal permite el uso de parámetros, la especificación con parámetros formales se caracteriza como *especificación parametrizada*. Si los parámetros formales se instancian con valores concretos o no existen, la especificación se denomina *especificación instanciada*. En las especificaciones pueden existir *opciones de implementación*, cuando la implementación puede tener determinadas características o no sin dejar de ser conforme con la especificación. En una Línea de Producto Software siempre hay opciones de implementación, puesto que existen diferentes variantes respecto de la especificación genérica, que son cada uno de los productos.

Una implementación es un conjunto compuesto por software y hardware que se comunica con su entorno y sus usuarios por medio unas determinadas interfaces. Una implementación es por tanto algo concreto y tangible, mientras que la especificación es algo abstracto. Por eso para poder relacionar la especificación con la implementación se precisa de una abstracción denominada modelo. Para una misma especificación puede haber diferentes modelos. En el caso del método propuesto en la Tesis Doctoral los modelos van a estar construidos con diagramas de estados jerárquicos (statecharts).

La conformidad entre una especificación y la implementación, denominada también implementación bajo prueba (en inglés IUT, *implementation under testing*), existe cuando la implementación es correcta respecto de la especificación. La conformidad se divide en dos partes:

Conformidad estática a nivel de características u opciones de implementación, si la especificación parametrizada está instanciada correctamente, es decir la combinación de opciones de implementación

de la implementación es válida. En el caso de una Línea de Producto Software, esta conformidad existe cuando las opciones de implementación elegidas resultan en un producto determinado e identificable.

Conformidad dinámica, consistente en que el comportamiento observable de la implementación es acorde con lo descrito por la especificación. Esta conformidad se caracteriza mediante la *relación de implementación* entre el modelo de la implementación y la especificación [ITU97], que en la Figura 3.2 se denomina *imp,*

$$imp \subseteq \text{MODELOS x ESPECIFICACIONES}$$

y el conjunto M_s de todos los posibles modelos de una especificación s se describe como

$$M_s = \{m \in MODELOS \mid m\ imp\ s\}$$

Por tanto, una implementación IUT es conforme dinámicamente respecto de una especificación s si su modelo asociado M_{iut} verifica que M_{iut} *imp* s.

Es decir, una especificación puede tener múltiples implementaciones conformes, y múltiples modelos conformes asociados. Se está postulando que siempre se puede elaborar un modelo válido de la implementación para probarla, lo que [ITU97] denomina *test assumption* (postulado de prueba).

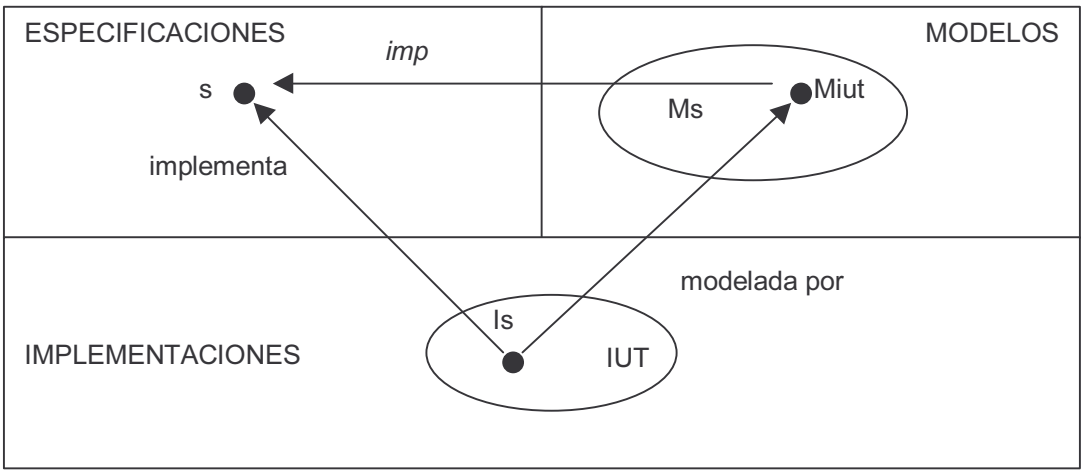


Figura 3.2 Relación entre Especificación, Modelo e Implementación

En el caso del método propuesto por la Tesis Doctoral el postulado de prueba o *test assumption* consiste en que se modela la especificación de requisitos mediante un conjunto de diagramas de estados jerárquicos y escenarios, cada uno de los cuales cubre una parte de los requisitos que constituyen la especificación. Si la implementación se comporta de acuerdo con lo descrito mediante las máquinas de estados, se concluye que la implementación bajo prueba es conforme desde el punto de vista dinámico. Para verificar la conformidad dinámica se ha escogido el algoritmo de prueba de máquina de estados denominado estrategia N+ en [Bin00] que se explica con más detalle en la parte de Estado de la Ciencia y en la parte de herramientas asociadas al método de la Tesis Doctoral.

[ITU97] dice que en general no es posible una certeza plena en cuanto a la conformidad debido por ejemplo, a posibles comportamientos no deterministas de la aplicación, restricciones en las posibilidades de control y observación de la implementación y el hecho práctico de que sólo es posible realizar un número finito de pruebas.

Un conjunto de pruebas puede ser según [ITU97] exhaustivo (toda implementación que lo pasa es conforme), robusto (toda implementación que no lo pasa es no conforme) o completo (exhaustivo y robusto a la vez). En la práctica los conjuntos de pruebas exhaustivos finitos no existen. El método de pruebas propuesto no busca conseguir conjuntos de pruebas de estas características, sino que busca optimizar en la práctica de la forma más estructurada posible el tiempo y los recursos disponibles para las pruebas.

3.2.2 Relación entre elementos de una Línea de Producto Software

Se profundiza ahora en la "relación" entre los componentes de una Línea de Producto Software, con la idea de expresar de un modo formal que dos productos P1 y P2 pertenecen a la misma familia. Para denominar esta relación, es bastante adecuado el concepto de analogía del diccionario de la Real Academia Española de la Lengua que define la analogía como "Relación de semejanza entre cosas distintas". Los miembros de una Línea de Producto Software son por tanto *análogos* entre sí, ya que comparten los "activos básicos" de la Línea de Producto Software y se diferencian en lo específico de cada producto.

La analogía entre miembros de una Línea de Producto Software se puede establecer en diferentes ámbitos, por ejemplo Arquitectura Software, Especificación de Requisitos, Plataformas Hardware comunes, etc. Desde el punto de vista del método propuesto en la Tesis Doctoral, la analogía a nivel de Arquitectura Software no es relevante, aunque de hecho exista, puesto que se está considerando la Línea de Producto Software desde el punto de vista de las pruebas de sistema donde se prueba con el enfoque de caja negra y por eso la estructura interna de la implementación no es lo más importante. Sin embrago la analogía a nivel de Especificación de Requisitos sí que es relevante para el método de pruebas de la Tesis Doctoral, porque el método parte de los requisitos para definir todo el proceso de pruebas y si existe una relación de analogía entre los requisitos de cada producto, los casos de prueba de cada producto van a tener también aspectos en común que permitirán optimizar el esfuerzo en las pruebas de la Línea de Producto Software.

Una posible forma de comprobar que la analogía entre dos productos existe, considerando solamente los requisitos, sería decir que, dados dos los conjuntos de requisitos *Pi* del Producto i y *Pk* del producto k

$$Pi=\{R1,R2,....Rn\}$$
$$Pk=\{r1,r2,....rm\}$$

hay analogía entre ambos si se verifica que

$$Pi \cap Pk \neq 0 \Rightarrow Pi \ \underline{analogo} \ Pk$$

Pero esto último no tiene por qué cumplirse, si los requisitos son muy detallados y numerosos, como suele ser habitual en los sistemas reales. Una coincidencia en una pequeña funcionalidad no implica una semejanza significativa que permita concluir que los dos productos software pertenecen a una Línea de Producto Software. Para que dos productos sean análogos, deben de coincidir en un número muy significativo de requisitos. El problema de esta afirmación es que no se puede expresar de manera formal. Por eso es más adecuado definir formalmente la analogía basándolo en requisitos de carácter global o características (*features*) porque se puede describir las diferencias entre productos de una manera más abstracta y breve.

Las características o *features* es lo que se usa desde el punto de vista práctico para comunicar las diferencias entre las variantes de la Línea de Producto y se definen previamente a la especificación de requisitos, que es el punto de partida del desarrollo software. Se dice por ejemplo, que se quiere un

coche con elevalunas eléctrico (nivel de características o *features*) en lugar de decir un coche en el que al pulsar un botón de mando situado cerca de la ventana arranque un motor que mueva los cristales de la ventana con una velocidad determinada hacia arriba o hacia abajo según se haya pulsado (nivel de requisitos). Ejemplos de caracterización de variantes mediante características o *features* en la literatura son [Ka02] y [Ka90].

En conclusión, es posible definir formalmente la analogía entre componentes de una Línea de Producto Software en el nivel de las características o *features*. En el nivel de Especificación de Requisitos de Sistema establecer de modo formal una relación de analogía no parece tener mucha utilidad práctica, ya que existen muchos más requisitos que características o *features* y se tiene que adoptar, en mi opinión, un criterio más pragmático de tipo cuantitativo del tipo de establecer que un porcentaje mayoritario (un 80%) de los requisitos de la Línea de Producto Software debe de ser genérico y el resto puede ser diferente para cada una de las variantes.

Siguiendo las ideas de [Ei02] se puede elaborar una tabla de características (*features*) a partir de la cual se construye un gráfico en forma de árbol cuyos nodos son los llamados conceptos, formados por pares de productos y propiedades. Esto tiene interés para abordar la planificación del desarrollo y las pruebas de varios productos en paralelo con propiedades comunes. Para simplificar, no se tiene en cuenta las propiedades que se excluyen mutuamente, ni la distinta importancia que pueda tener una propiedad frente a otra.

El *mapa de producto* contiene todas las características de cada producto y se puede definir desde el punto de vista formal como una relación binaria $M \subseteq \Pi \times C$ entre un conjunto de productos Π y un conjunto de características C. La siguiente tabla es un ejemplo simple de mapa de producto:

	Características						
Productos	almendras	nata	whisky	helado	chocolate	bizcocho	guindas
tarta reina	X	x	x			x	
tarta especial	X		x	x		x	
tarta suprema	X			x	X	x	x

Tabla 3.1 Ejemplo de mapa de producto

Para un mapa de producto dado, las *características comunes* de un cierto conjunto de productos $P \subseteq \Pi$ se define como $\sigma(P)$:

$$\sigma(P)=\{c \in C \mid (p,c) \in M \ \forall \ p \in \Pi\}$$

En el ejemplo, las características comunes de *tarta reina* y *tarta especial* son *bizcocho, almendras,whisky*.

Análogamente se define formalmente el conjunto $\tau(K)$ de *productos comunes* dotado de un conjunto de características $K \subseteq C$:

$$\tau(K)=\{p \in \Pi \mid (p,c) \in M \ \forall \ c \in C\}$$

En el ejemplo los productos que tienen en común la característica *whisky* son *tarta reina* y *tarta especial*.

Para poder manejar desde un punto de vista formal los productos con características comunes, se define el *concepto* como una combinación de productos y características tales que todos los productos tienen todas las características y todas las características son comunes a todos los productos.

$$c=(P,K) \text{ si y solo si } K=\sigma(P) \text{ y } P=\tau(K)$$

De forma intuitiva, el concepto sería en la tabla del ejemplo, un rectángulo con todas las casillas rellenas y donde las permutaciones de filas y columnas están permitidas. Volviendo al ejemplo, dos posibles conceptos son C1=({ *tarta especial, tarta suprema*},{ *bizcocho, almendras, helado* }) y C2=({*tarta reina, tarta especial, tarta suprema*},{*bizcocho, almendras*}). Comparando C1 y C2, se observa que los productos de C1 son un subconjunto de los productos de C2 y que las características de C2 son, a su vez, un subconjunto de las características de C1, por lo que se puede decir que C2 es más general que C1. Se puede representar la relación entre C1 y C2 con el signo ≤ de tal forma que:

$$C1=(P1,K1) \leq C2=(P2,K2) \Leftrightarrow P1 \subseteq P2 \text{ y } K2 \subseteq K1$$

Hay conceptos que no se pueden comparar usando la relación ≤. Por ejemplo entre C5=({*tarta suprema*},{ *bizcocho, almendras, helado, chocolate, guindas*}) y C1=({ *tarta especial, tarta reina*},{ *bizcocho, almendras, whisky* }), tanto C1 ≤ C5 como C5 ≤ C1 es falso, pues ni P1 ⊆ P5 ni P5 ⊆ P1.

Basándose en esta relación "≤.", [Ei02] propone construir con los conceptos un grafo acíclico cuyos nodos son los conceptos y una flecha une dos nodos cualesquiera Cx y Cz cuando Cx ≤ Cz. El grafo permite identificar al concepto más general de todos, pues es aquel del que sólo salen flechas y no entra ninguna. El problema que plantea [Ei02] se puede resolver también de manera no formal, inspeccionando la tabla de características manualmente o mediante un programa si la tabla es muy grande.

En conclusión, se define que los miembros de la Línea de Produto Software son *análogos* cuando existe un *concepto* general distinto del conjunto vacío que contiene las características comunes de la Líneas de Productos Software:

$$\exists\, C_x=(P_x,K_x) \wedge C_x \neq \varnothing \mid \forall\, C_i=(P_i,K_i) \neq C_x \Rightarrow C_i \leq C_x$$

En sistemas reales no es trivial elaborar la tabla de características que en [Ei02] se toma como base para representar el mapa de producto, debido a que se precisa de un gran conocimiento del dominio para describir las características con el grado de abstracción adecuado y no todas las partes implicadas en una línea de productos (gestores, usuarios, desarrolladores, analistas funcionales, responsables de pruebas, etc.) tienen ese conocimiento.

3.2.3 Trazabilidad

Se dice que hay trazabilidad entre dos elementos asociados, si a partir de uno de ellos se puede inferir el otro y viceversa. Para que se garantice la trazabilidad entre requisitos y casos de prueba (trazabilidad horizontal) se van a estructurar los requisitos de forma que las diferentes variaciones dentro de la línea de productos aparezcan de forma explícita y a partir de los requisitos, se confeccionarán diagramas de estados jerárquicos de los que se derivan casos de prueba. La trazabilidad entre casos de prueba y elementos de la arquitectura (trazabilidad vertical) no es un objetivo prioritario en este método.

La Figura 3.3 muestra una panorámica global o metamodelo de lo que se pretende conseguir, que es una interrelación entre requisitos, pruebas y arquitectura, articulada a través de dos niveles o tipos de trazabilidad "horizontal":

- Pruebas-Requisitos. Qué requisitos se han probado.
- Pruebas-Arquitectura: Qué partes de arquitectura se han probado.

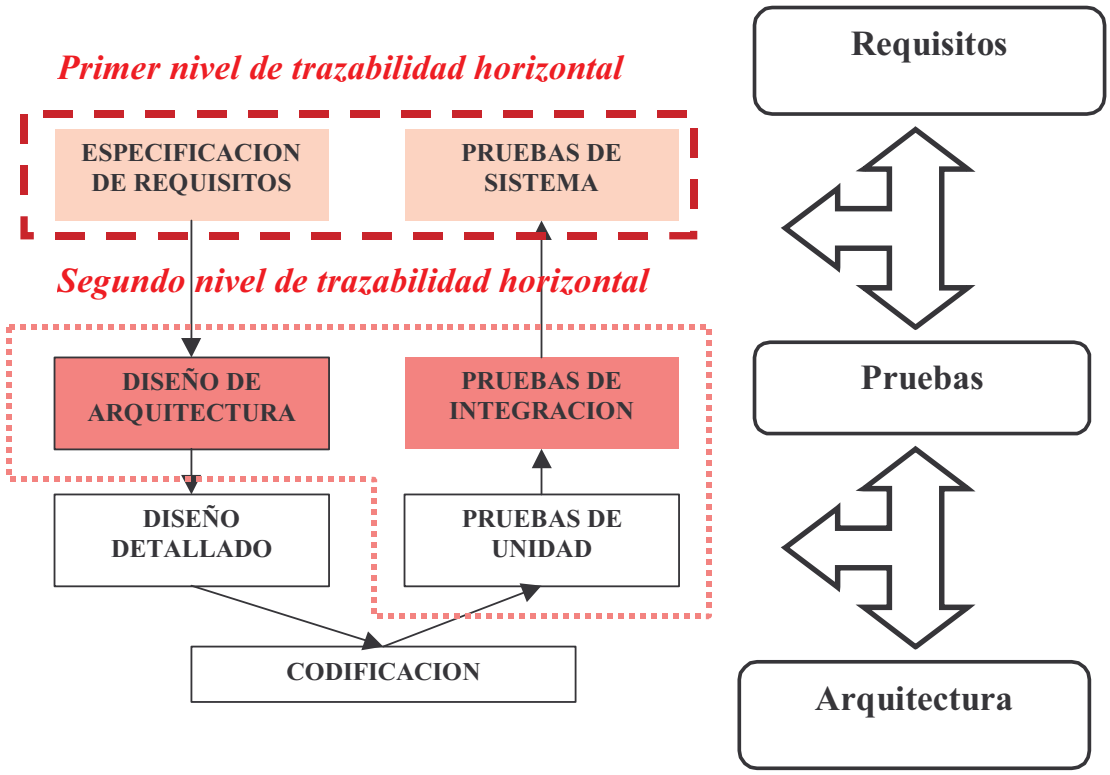


Figura 3.3 Metamodelo de la relación Requisitos-Pruebas-Arquitectura

El primer nivel de trazabilidad abarca los requisitos y los casos de prueba y tiene un alto grado de detalle, pues se busca que cada requisito tenga casos de prueba asociados y viceversa con el objetivo de lograr medir la cobertura de las pruebas. La métrica de cobertura para este nivel de trazabilidad es el porcentaje de requisitos del sistema comprobados, pero debe de tomarse como un dato de carácter relativo, ya que puede suceder que la métrica indique un grado de cobertura que no se ajuste a la realidad en determinadas circunstancias (un número pequeño de requisitos especifica una función compleja o que requiere mucho esfuerzo de preparación del entorno de pruebas, la especificación contiene requisitos incompletos, etc).

El segundo nivel engloba la relación entre los casos de prueba de integración y las pruebas unitarias con la arquitectura. No se precisa un método de descripción de arquitectura en particular, si bien es recomendable seguir algún método estándar, como el IEEE P1471 y una notación de amplia difusión como UML.

Este segundo nivel de trazabilidad se va a ocupar de qué interfaces y que módulos de la arquitectura se comprueban mediante cada caso de prueba. La medida de cobertura que proporciona esta trazabilidad es también significativa, y complementaria a la medida que proporciona el primer nivel de trazabilidad. Es claro que las pruebas de integración no pueden dejar de lado ninguna de las interfaces del sistema de manera injustificada. Por otra parte, realizar pruebas unitarias de todos los módulos de una arquitectura indica un nivel de pruebas muy exhaustivo, pero que en determinadas circunstancias no va a ser posible por falta de recursos. El hecho de que las pruebas de integración o las unitarias no sean muy exhaustivas no indica que las pruebas sean deficientes, mientras que decir que sólo se

prueba la tercera parte de los requisitos sí que lo revela. La tabla que hay a continuación resume lo dicho.

NIVEL DE TRAZABILIDAD	Métricas de Cobertura	Importancia	Grado de Detalle
Requisitos con casos de prueba	Porcentaje de requisitos del sistema comprobados	ALTA	ALTO
Arquitectura con pruebas unitarias y de integración	Porcentaje de Interfaces comprobadas Porcentaje de Módulos comprobados Porcentaje de código ejecutado en las pruebas unitarias de cada módulo	MEDIA	MEDIO/ALTO (en función de los recursos disponibles)

Tabla 3.2 Niveles de trazabilidad

El método va a incorporar automatización de pruebas, por las razones que se han indicado anteriormente, de la necesidad de un esfuerzo de pruebas de regresión en las Líneas de Producto Software. Para poder automatizar las pruebas necesitamos un modelo del sistema que cumpla los siguientes criterios:

- El modelo refleja adecuadamente la implementación que prueba. Representa todo lo que hay que probar.
- El modelo no contiene detalles irrelevantes que hacen las pruebas más costosas en tiempo y en recursos.
- Conserva el grado de detalle necesario para detectar fallos y demostrar que se cumplen los requisitos del sistema.
- Representa todos los eventos ante los que el sistema debe responder. Los eventos se generan con un mensaje, una excepción, o una interrupción.
- Se puede saber qué acciones han sido ejecutadas.
- Se puede comprobar el valor del estado del sistema automáticamente.

Un sistema modelado mediante diagramas de transición de estados cumple estas características, y por eso va a emplearse esta técnica como base de los modelos de prueba.

3.3 Elementos del Método de Pruebas

3.3.1 Requisitos de Líneas de Producto Software

Al hablar de la trazabilidad se ha explicado la importancia de los Requisitos en el método de pruebas propuesto. A continuación se detalla la estructura de los requisitos de una Línea de Producto Software, desglosándolo, como indica la Figura 3.4, en dos niveles: un primer nivel “grosso modo” de requisitos globales o características, que se emplean en las actividades de planificación y gestión de la Línea de Productos Software (seguimiento del proyecto, asignación de recursos, etc.), y un segundo nivel más detallado, formado por los Requisitos de Sistema y las variantes en dichos requisitos, que constituyen la Especificación de Requisitos de sistema o SRS (*System Requirements Specification*) que se los emplea como punto de partida del desarrollo de software.

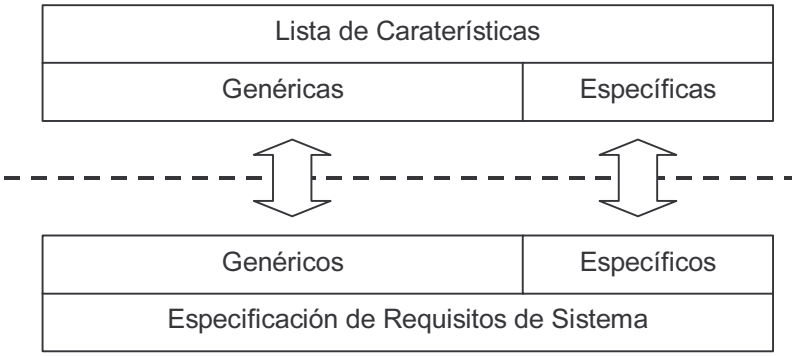


Figura 3.4 Niveles de detalle en los Requisitos de Línea de Producto Software

La trazabilidad, simbolizada por las flechas en la Figura 3.4, entre Lista de Características y Especificación de Requisitos de Sistema consiste en asociar los requisitos a las distintas características, lo que puede hacerse por requisitos individualmente, o por grupos de requisitos. Por ejemplo, si se emplean Casos de Uso para organizar los requisitos, se puede englobar uno o varios casos de uso en una característica como sugiere [Go01]. Es una labor que debe de realizarse de forma manual y que puede suponer bastante esfuerzo. Desde un punto de vista pragmático, es más fácil hacerlo desde las *features* a los requisitos, que a la inversa, ya que por definición hay muchos mas requisitos que *features*.

3.3.1.1 Nivel de Requisitos Globales

Para definir el primer nivel de requisitos, se va a partir de [CeR03] donde se expone un modelo que podría denominarse genérico de Línea de Producto Software, siguiendo las ideas del estándar IEEE 1471-2000 y que se muestra en la Figura 3.5. La Línea de Productos Software tiene al menos una misión u objetivo, de la misma manera que cada uno de los productos. En este sentido, la Línea de Productos Software debe de ser entendida como un medio para facilitar el desarrollo del producto, dentro de un cierto entorno y una determinada organización. Dentro de la organización existen unas personas denominadas agentes en el modelo de la Figura 3.5, que influyen sobre la Línea de Productos Software ya sea directamente p.ej. tomando decisiones, o indirectamente estableciendo requisitos, etc.

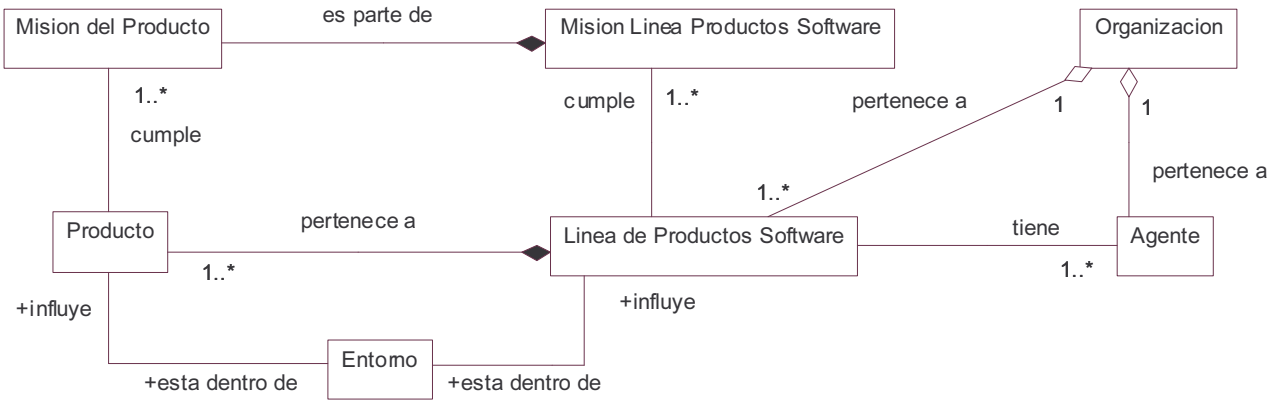


Figura 3.5 Modelo genérico de Línea de Producto Software

En la Figura 3.6 se muestra el modelo de características o *features* de Línea de Productos Software que se ha elaborado basándose en el de [CeR03]. La Línea de Productos Software se describe mediante un abanico de características de las cuales, algunas están presentes en todos los productos, las características genéricas, mientras que el resto son las características específicas, que diferencian los productos unos de otros.

Dentro de los distintos Agentes de la Línea de Productos Software, el responsable es quien decide, con el asesoramiento de otras personas de la organización si fuera necesario, si se incluye en la Línea de Productos Software una cierta característica o no. La labor del responsable puede crecer en complejidad dependiendo de la forma de gestión de la Línea de Productos Software. Si la Línea de Producto Software es simplemente una fusión de varios sistemas software existentes que se toman tal como son sin cambiar nada, es más sencilla, si por el contrario debe de decidir qué características se incorporan a la Línea de Producto Software en función del entorno externo de la organización, la situación es muy complicada, porque primero tiene que definir con la máxima precisión posible en qué consiste la nueva característica, y luego determinar si es conveniente incorporarla o no considerando su complejidad, los recursos existentes en la organización para implementarla, si es reutilizable para varios productos o no, etc.

El Responsable debe de negociar con el cliente, que es el que solicita que se incluyan características en la Línea de Producto Software, y que puede ser externo a la organización o interno. El responsable define las características a partir de lo que le comunican los clientes (entiende lo que el cliente pide) y puede darse el caso de que haga propuestas alternativas al cliente cuando ve que lo que el cliente pide puede solucionarse de otra forma menos costosa en esfuerzo, utilizando por alguna característica ya existente en la Línea de Producto Software.

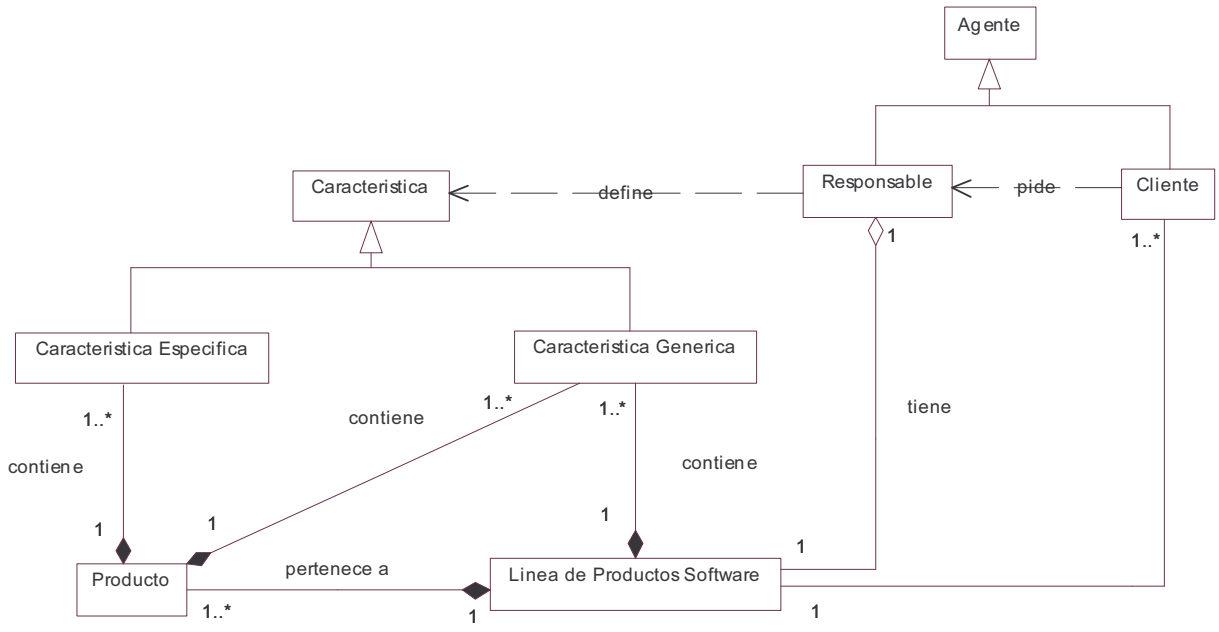


Figura 3.6 Modelo de características de Línea de Producto Software derivado del modelo genérico

El modelo de características tiene un grado de detalle que es suficiente desde el punto de vista del método de pruebas propuesto en la Tesis Doctoral, pero que es menor que el propuesto por otros autores como [Ka02], en el que se estructuran características en forma jerárquica y se definen además varias capas (características funcionales, entorno, tecnología del dominio, tecnología de implementación) en la jerarquía para agrupar las características. Evidentemente, un mayor grado de

estructuración en las características o *features* siempre va a ser algo positivo, pues permite hacerse una idea de todo lo que tiene que tener la Línea de Producto Software más rápidamente y localizar sus partes genéricas a través de las relaciones que hay entre las características.

Una característica o *feature* está detallada por múltiples requisitos y un requisito puede estar asociado a una o más características, pero lógicamente hay muchos más requisitos que características. Esto quiere decir que se pueden agrupar los requisitos por características con las que están asociados. Puesto que de los requisitos se van a derivar los casos de prueba, se puede aplicar la estructura que dan a los requisitos las características o *features* a los casos de prueba.

3.3.1.2 Nivel de Requisitos de Detalle

Para modelar el nivel de requisitos de detalle se va a partir de las aportaciones de [CeV03] y [Lut01], organizan en forma jerárquica los requisitos de la Línea de Producto Software.

[Lut01] (ver Figura 3.7) clasifica los requisitos de la Línea de Producto Software en Requisitos Comunes y Requisitos Específicos, que son los que describen las variaciones, y que se dividen en Opciones, Requisitos que implican a su vez otros requisitos y Requisitos excluyentes. El Requisito Específico tiene los siguientes atributos:

- Causa: Motivo de la variabilidad.
- Momento de Resolución: cuándo se establece la variabilidad (compilación, enlace, instalación, ejecución)
- Mecanismo de Variabilidad: cómo se establece la variabilidad (parametrización estática o dinámica, etc.).

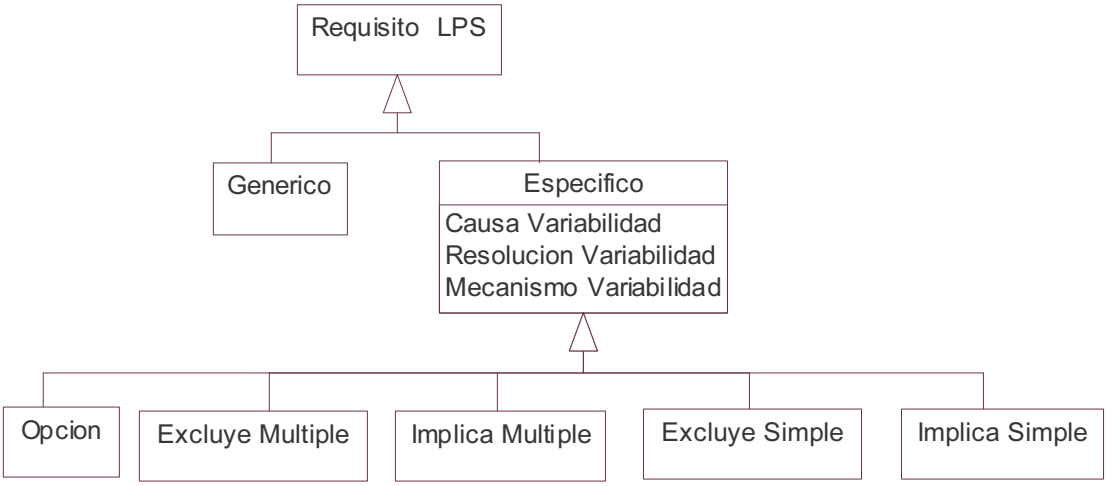


Figura 3.7 Requisitos genéricos y específicos de Línea de Producto Software en [Lut01]

Desde el punto de vista de la especificación de requisitos, no está clara la utilidad de los dos últimos atributos definidos por [Lut01], pues mezclan en los requisitos información propia del diseño. [CeV03] es más general, pues no distingue entre requisitos comunes y específicos, sino que define, como se muestra en la Figura 3.8, para cada requisito una relación del requisito con otros cero o más requisitos, que puede ser del tipo:

- Relación lógica: Exclusión, Coexistencia, Obligatoriedad, Alternativa.
- Relaciones jerárquica: Generalización, Especialización.
- Relación de evolución: Revisión.

La relación de requisito puede existir según [CeV03] en los requisitos funcionales y en los no funcionales, que [CeV03] clasifica conforme al estándar [ISO91].

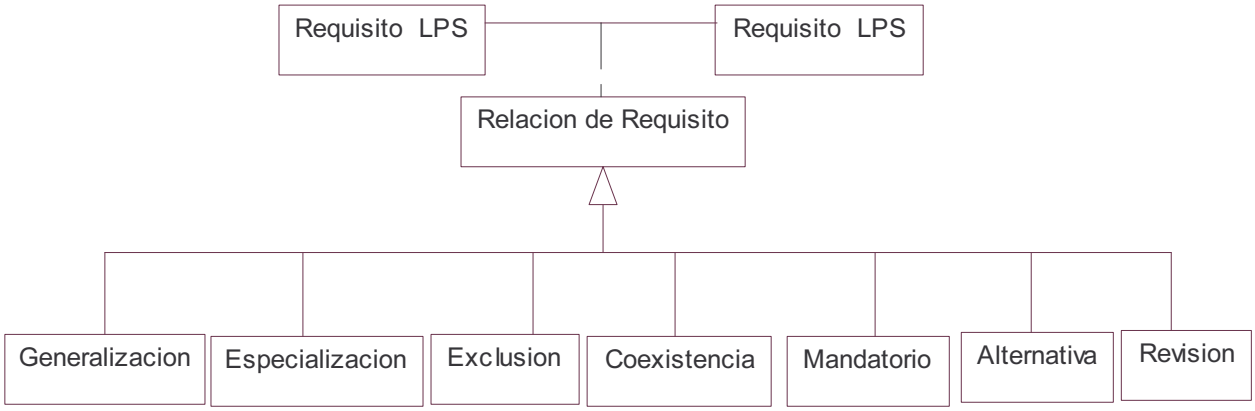


Figura 3.8 Relación entre requisitos de Línea de Producto Software

A partir de estos dos modelos de requisitos genéricos se ha elaborado para el método de pruebas un modelo de requisitos de línea de producto aplicado, realizando las siguientes suposiciones de carácter práctico:

- Lo más habitual es que la Línea de Productos software sea fruto de la integración de varios proyectos similares que discurren más o menos paralelamente. Como intenta mostrar la Figura 3.1, los requisitos comienzan siendo específicos de un proyecto, que es lo mismo que decir un producto determinado, y tras una fase de análisis, una parte de ellos se incorpora al conjunto de requisitos genéricos. Un posible criterio para la fase de análisis podría ser el incorporar en el producto genérico un ochenta por ciento de la funcionalidad y en los productos específicos el veinte por ciento restante, pues si la parte genérica no es la que predomina, no tiene mucho sentido hablar de Línea de Producto Software, sino de sistemas individuales. Por lo tanto, un atributo importante del requisito será si es genérico o específico, y en este último caso, el producto al que pertenece.

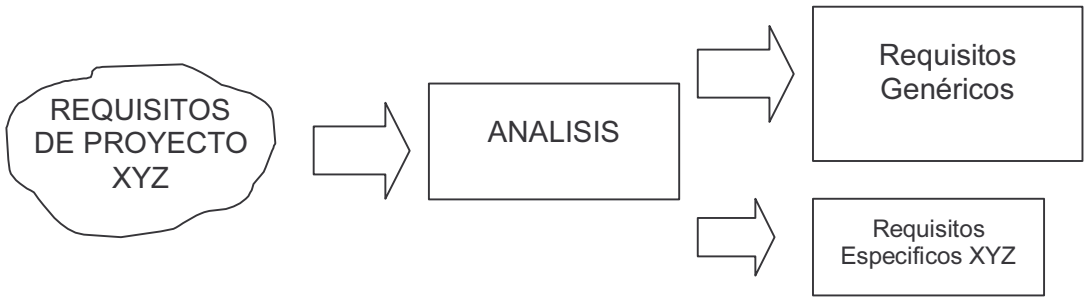


Figura 3.9 Caracterización de Requisitos en Genéricos y Específicos

- El método de pruebas va a construir para las pruebas de sistema modelos de máquinas de estados a partir de los requisitos y los requisitos que se van a usar para ello son fundamentalmente los requisitos funcionales. En la medida en que describen un comportamiento observable del sistema, se pueden modelar con una máquina de estados. Por eso, a diferencia de [CeV03], que clasifica los

requisitos de forma más detallada, desde el punto de vista del método es suficiente clasificar los requisitos en funcionales y no funcionales, aplicando el método de pruebas sobre los primeros y verificando los segundos mediante los procedimientos que apliquen en cada caso, que habitualmente no encajarán en el ámbito del método de pruebas.

- El número de requisitos del sistema se estima por encima del millar, considerando que cada requisito está expresado en dos o tres frases, es decir, que se trata de un sistema de una complejidad relativamente alta, ya que si no, no se justifica el esfuerzo de implementar una Línea de Producto Software. Por otro lado, no se considera la posibilidad de englobar en un requisito una gran cantidad de información (3 páginas de texto, por ejemplo), porque como ya se ha dicho, se busca demostrar que los casos de prueba cubren todos los requisitos y eso no es factible si cada requisito abarca demasiada funcionalidad.

- Se está presuponiendo también que los requisitos están gestionados mediante una base de datos o herramienta similar, y que existe un proceso de revisión y de gestión de configuración de los requisitos, puesto que no tendría sentido que se implementara un proceso de pruebas riguroso como el que se describe en la Tesis Doctoral si los requisitos están fuera de control, o son ambiguos e incompletos y sería muy difícil construir a partir de esos requisitos modelos formales o semiformales.

- Se ha establecido una división de los requisitos en requisitos de detalle y requisitos globales o características. Si el número de requisitos fuera muy elevado, para reducir la complejidad se puede clasificar los requisitos de detalle en "niveles jerárquicos de especificación", estableciendo un primer nivel con los requisitos más "*esenciales*", y uno o más niveles con el resto de requisitos, de forma que todo requisito o es un requisito esencial o está ligado a un requisito esencial. Establecer esta jerarquía sirve para que cuando se modele un conjunto de requisitos mediante una máquina de estados, se pueda reducir el número de requisitos a tener en cuenta considerando únicamente los requisitos *esenciales*.

- Una distinción importante en sistemas donde la variedad está sobre todo ligada a la multiplicidad de plataformas hardware o software es el distinguir entre un conjunto de requisitos independientes de la plataforma y otro conjunto de requisitos dependientes de la plataforma, análogamente al *Platform Independent Model (PIM)* y el *Platform Specific Model (PSM)* que incluye el OMG (Object Management Group) en su Model Driven Architecture (MDA) [OMG03][He03][Mel04]. Esta distinción no es tan importante en sistemas de tiempo real que usan soluciones hardware fijas empotradas, como algunos de los sistemas de control de tráfico ferroviario en los que tiene experiencia el autor de la Tesis Doctoral, en los cuales la plataforma hardware requiere también una certificación oficial, lo que limita el número de plataformas hardware a emplear.

Teniendo en cuenta todas estos factores, se va a definir el Requisito de Línea de Producto Software como un conjunto compuesto por los siguientes elementos:

$$Req\_LPS = \{ID, Texto, Categoría, Estado, Versión, Tipo\_Req, Producto\}$$

- ID: identificador del Requisito.
- Texto: contenido del Requisito.
- Estado: Válido, En revisión, Borrado, Nuevo.
- Categoría: Esencial (para distinguir los requisitos esenciales si se usan), Requisito, Epígrafe, Figura, Comentario.
- Versión de la especificación a la que pertenece el Requisito. Tienen que estar definidas unas versiones de la SRS asociadas con las versiones de los productos.
- Tipo_Req: Tipo de Requisito. Este campo es opcional y sus valores dependerán del dominio de aplicación. Por ejemplo, en el caso de los sistemas ferroviarios es interesante diferenciar los

requisitos de seguridad, pues se examinan con más detalle en las pruebas y en la certificación del sistema.

- Producto al que pertenece el Requisito. Si el requisito es genérico, este campo no existe.

La Figura 3.10 representa el requisito en forma de diagrama de clase.

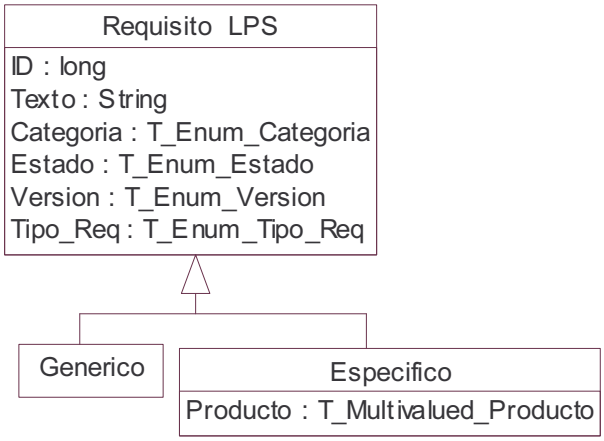


Figura 3.10 Diagrama de Clase de Requisito de Línea de producto Software

No se ha incluido como en [CeV03] y [Lut01] en el requisito información de relación frente a otros requisitos, es decir, si excluye otros requisitos o por el contrario es inseparable respecto de otros requisitos. La razón es que esta información, si bien es útil, es luego costosa de mantener a priori, salvo que se tenga el soporte de herramientas software adecuado. Lo que no va a hacer ninguna herramienta software es el trabajo de definir la relación entre requisitos, pues para hacer esto se requiere un conocimiento profundo del dominio de aplicación y es una tarea que tendrá que hacerse por los analistas de requisitos manualmente. Lo que sí varía de unas herramientas a otras es la capacidad de visualización de las relaciones entre requisitos. La herramienta de gestión de requisitos con la que tiene experiencia el autor de la Tesis Doctoral, DOORS de Telelogic, se puede adaptar escribiendo los "scripts" oportunos para que refleje estas relaciones de algún modo, ya sea gráfico o textual. [We03] advierte del peligro de definir trazas entre requisitos alegremente sin considerar que luego hay que mantenerlas. Si el número de requisitos es muy elevado, mantener ese entramado de relaciones puede llegar a ser muy costoso, y además, no parece de mucha utilidad práctica. Sí que lo es en el nivel de requisitos globales o características, pues para planificar el desarrollo de una cierta variante es importante saber que una determinada característica excluye a otra o la implica.

3.3.1.3 Requisitos de Tiempo Real

En los objetivos de la Tesis Doctoral se ha dicho que se quiere elaborar un método de pruebas para sistemas de tiempo real. Los lectores perspicaces habrán advertido que hasta ahora esto no se ha mencionado en este capítulo de la Tesis Doctoral, que es el que describe el método de pruebas. Es el momento de abordar esta cuestión, determinado en qué medida se ven afectados los requisitos y los casos de pruebas cuando corresponden a un sistema de tiempo real.

En los sistemas de tiempo real algunos de los requisitos van a ser plazos críticos. Si el sistema de tiempo de real es una línea de Producto Software, puede haber diferencias en los plazos críticos de los diferentes productos.

Aspectos típicos en los sistemas de tiempo real son, entre otros: Sensores externos, concurrencia, semáforos, regiones criticas; colas de mensajes, excepciones, temporizadores, entrada/salida, planificación de tareas [Th01] y gestión de memoria. Estos aspectos no afectan a los requisitos directamente, sino que son una consecuencia de los mismos: se usa un determinado algoritmo de planificación de tareas, o una cierta gestión de memoria, etc. con el objetivo de satisfacer unos ciertos requisitos.

Dado que en el modelo de requisitos que se propone en el método (ver Figura 3.10) no se expresan mediante un lenguaje matemático, ni mediante lógica temporal, sino que se usa simplemente texto, no tiene repercusión en el requisito si es un plazo crítico o no. Se podría utilizar, si fuera necesario, un valor especial en el campo *Tipo_Req* (Tipo de Requisito) para diferenciar los requisitos que sean plazos críticos del sistema de los demás.

Puede haber casos en los que sea insuficiente una descripción textual para definir un requisito de tiempo real, porque haya que precisar no sólo el plazo sino que se exija definir de forma cuantitativa la desviación admisible respecto de dicho plazo crítico en el tiempo de reacción del sistema. Para superar esta limitación, puede ser útil el concepto de función de utilidad que caracteriza un plazo crítico descrito en [Dou99] y que está detallado en la Tabla 3.3.

Función de Utilidad	Descripción
Unaria	U(t) = 1 cuando se termina la acción si t < plazo t0 y si no, U(t) =0 (es una función tipo escalón)
Lineal	U(t) = at + b, donde a y b son constantes si t < t0 y U(t) = c , donde c es otra constante si t > t0
General	U(t) = f(t)
Progresiva	U(t,w) ,siendo w el porcentaje de acción que se ha finalizado.

Tabla 3.3 Tipos de Funciones de Utilidad

El modelo de requisitos que se ha expuesto previamente, en caso de que se necesitara incluir las funciones de utilidad en algunos de los requisitos, quedaría como muestra la Figura 3.11, con una función de utilidad asociada al requisito que lo necesite, ya sea específico o genérico. Este grado de refinamiento no tiene impacto en el nivel de requisitos globales o características y sólo es relevante en el nivel de requisitos de detalle, propio de la SRS (Especificación de Requisitos de Sistema).

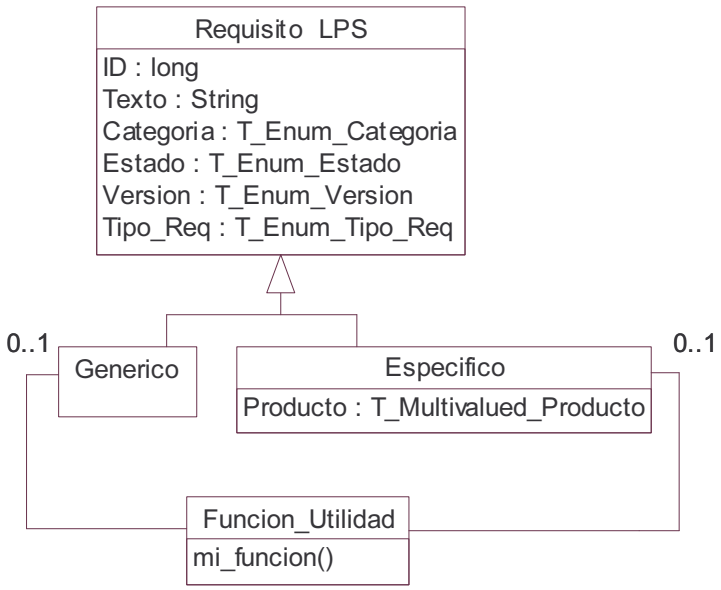


Figura 3.11 Modelo de requisitos de Línea de Producto Software extendido para sistemas de tiempo real

Lo habitual en los sistemas de control del tráfico ferroviario entre los que se ha elegido el caso de aplicación de la Tesis Doctoral, es que los requisitos temporales se expresen en lenguaje natural, y que las funciones de utilidad no sean necesarias. Lo que sí se hace en los sistemas ferroviarios cuando se desarrolla conforme al estándar CENELEC es un análisis de árbol de fallos tipo FMEA [Lev95] para determinar el grado de probabilidad de que se produzca una situación de riesgo, asumiendo unos tiempos de reacción determinados en el sistema. Si el análisis FMEA concluye que con un determinado plazo no se está debajo del nivel de probabilidad de fallo exigido por la norma, hay que redefinir el requisito y establecer un plazo crítico menor.

3.3.2 Clases de Prueba, Casos de Prueba y Datos de Prueba

Al principio de este apartado metodológico de la Tesis Doctoral se ha propuesto para las Líneas de Producto Software un modelo de ciclo de vida basado en el clásico modelo en V. Hasta ahora se ha explicado el lado "izquierdo" de la V (especificación de requisitos) y ahora se explica el lado "derecho" (especificación y realización de pruebas). Los aspectos de diseño e implementación no son objeto de interés desde el punto de vista del método, ya que se ha circunscrito su aplicación a las pruebas de sistema, donde la estructura interna del software no es visible.

Los *requisitos* se clasifican en diferentes categorías funcionales, que se van a denominar *clases de prueba*. El criterio para agrupar un conjunto de requisitos en una clase de prueba es si están relacionados con el comportamiento de una entidad relevante del sistema. Esa entidad es un objeto del dominio cuyo comportamiento se puede modelar mediante un diagrama de estados jerárquico. Estas identidades relevantes del sistema se identifican en la fase de análisis de dominio, previa a la especificación de requisitos, como muestra el modelo de ciclo de vida software de la Figura 3.1. Por ejemplo, en un sistema bancario, una clase de prueba puede ser el objeto cuenta corriente, con un diagrama de estados que modele su comportamiento.

Es en este paso del método donde tiene lugar el llamado postulado de prueba o *test assumption*, ya que se establece un modelo de la especificación, en este caso semiformal y se toma como base para definir los casos de prueba. La *clase de prueba* tiene un grupo de *casos de prueba* relacionados, que se

derivan de probar las transiciones de la *máquina de estados* que modela la clase de prueba mediante una cierta estrategia.

Se definen también *clases de prueba* para los requisitos de aspectos tales como calidad, seguridad, rendimiento y fiabilidad, que afectan a todo el sistema y no se pueden encasillar dentro de una funcionalidad concreta. Estas *clases de prueba* no van a automatizarse.

Un caso de prueba, según se muestra en la Figura 3.19 está definido por la n-tupla:

$$CP=\{Ei,e,Ef,Rj1,...,Rjm\}$$

Donde *CP* es un caso de prueba dado, *Ei* es el estado inicial, *Ef* el estado final, *Rj1,...,Rjm* son los m requisitos asociados a la transición que produce el evento *e.*

Aquí se está asumiendo que la estrategia de pruebas de la máquina de estados es recorrer todas las transiciones existentes entre estados y que no hay casos de prueba que incluyan estados intermedios. Esta estrategia, que ha sido la elegida en la parte de Aplicación Práctica de la Tesis Doctoral, se ha empleado considerando que su simplicidad facilita la automatización y que se va a tener un número de casos de prueba elevado, si bien como se ha dicho en la parte de estado de la ciencia (ver apartado en página 59, sobre estrategias de prueba) existen estrategias de prueba de máquinas de estados con mayor capacidad de detección de errores.

Para incrementar la capacidad de detección de fallos, utilizando una estrategia de pruebas N+ (ver apartado en página 59, sobre estrategias de prueba), se define un tipo de caso de prueba denominado *recorrido de prueba*, que engloba varios casos de prueba que se ejecutan secuencialmente:

$$RP=\{CP_0,CP_1,...,CP_n\}$$

tal que el estado final de cada caso de prueba es el inicial del caso de prueba siguiente

$$\forall CP_i \in RP,\ i=0...n \Leftrightarrow CP_j=\{Ei,e,Ef,...\} \wedge CP_{j+1}=\{ Ef,b,Ef',...\}$$

Cada dato de prueba *DP*, que es la implementación correspondiente a un caso de prueba, se define de la siguiente forma:

$$DP=\{CP,\ Pon\_Estado(Ei),\ Manda\_Mensaje(e),\ Lee\_Estado(Ef),\ Oid\}$$

Donde *CP* es el caso de prueba que se quiere implementar y *Oid* es el identificador del objeto en el sistema que se escoge para realizar las pruebas. Las funciones *Pon_Estado(Ei), Manda_Mensaje(e)* y *Lee_Estado(Ef)* transforman el valor que se define en el caso de prueba en una llamada al sistema, que variará en función de la implementación. Lógicamente, se está asumiendo que los objetos de una clase de pruebas dada, se comportan todos de igual manera con independencia del número identificador que se les asigne a cada uno.

Se está asumiendo que en la línea de productos en la que se va a aplicar este proceso de prueba, existen en los activos básicos las funciones (o mecanismo similar) siguientes:

- Acceso al valor del estado de los objetos (lectura y escritura).
- Envío de eventos para disparar las transiciones de los diagramas de estados.
- Servicio de medición de tiempos.
- Envío de mensajes y monitorización de los mensajes de respuesta para las pruebas de los escenarios.

El disponer de estas funciones permitirá automatizar las pruebas, pues podrán invocarse desde los correspondientes guiones (scripts) de prueba.

3.3.2.1 Notación de Prueba

[ITU97] llama *notación de prueba* al lenguaje formal en el que se expresa un caso de prueba. Asumiendo, según lo dicho en el apartado 3.3.2, que un caso de prueba CP está definido de la siguiente forma

$$CP=\{Ei,e,Ef,Rj1,...,Rjm\}$$

en la que los elementos *Ei,e,Ef* (Estado incial, evento y estado final) son obligatorios y los elementos *Rj1,...,Rjm* (requisitos asociados a la transición) son opcionales.

Se puede expresar el caso de prueba con la notación EBNF (Formalismo Extendido de Backus-Naur) de la forma siguiente:

$$CP=EieEf\{R\}$$

O también, en la línea de [Mor02] y [ESI02], usando el lenguaje XML, con el que quedaría de la siguiente manera:

```
<xs:element name="Caso de Prueba">
<xs:complexType>
     <xs:attribute name="Estado Inicial" type="xs:string"/>
     <xs:attribute name="Evento" type="xs:string"/>
     <xs:attribute name="Estado Final" type="xs:string"/>
     <xs:attribute  name="Requisito"  type="xs:string"  minOccurs="1"
     maxOccurs="unbounded"/>
</xs:complexType>
</xs:element>
```

El recorrido de prueba se expresaría así:

$$RP=\{CP\}$$

3.3.3 Diagramas de estados

Los diagramas de estados (statecharts) describen el comportamiento de los elementos del sistema. Una transición está definida por el estado inicial, el estado final y el evento que produce la transición, y tiene una serie de requisitos asociados. La Figura 3.12 muestra un ejemplo de todos los casos de prueba que se pueden formular sobre un diagrama de estados sencillo. Esto permite hacer ya un cálculo de cobertura de requisitos de las pruebas. Si cada transición es un caso de prueba, probando todas las transiciones estamos probando un determinado número de requisitos, ya sean genéricos o específicos.

Los diagramas de estado se elaboran a partir de la información contenida en los requisitos. Esta tarea debe de realizarse con ayuda de expertos del dominio que colaboren en verificar que los modelos elaborados son completos y no tienen errores.

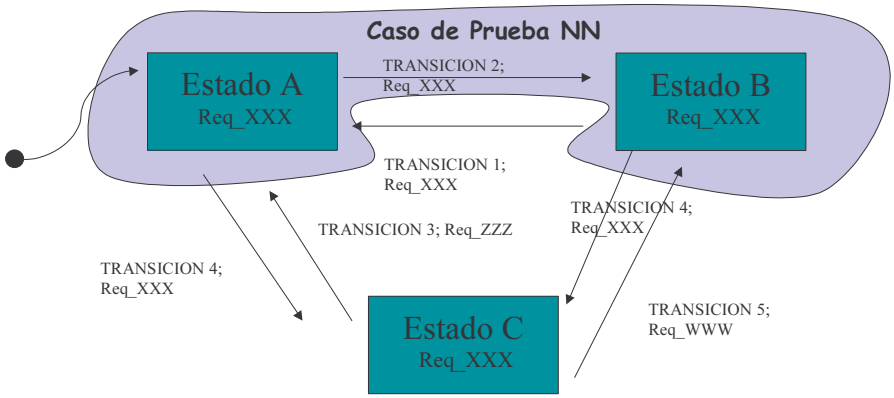


Figura 3.12 Diagrama de estados con información de requisitos

Dependiendo de la aplicación que se trate, es posible, que así como se ha distinguido entre requisitos genéricos a toda la línea de productos y requisitos específicos, existan también diagramas de estados genéricos que tengan instancias diferentes en varios miembros de la familia de productos (Figura 3.13). Se aplica el mismo concepto de asociar los requisitos con las transiciones en los diagramas de estados genéricos y en los específicos. El diagrama de estados específico no puede eliminar transiciones, ni borrar estados, ni añadir nuevos supra-estados que no existan en el diagrama de estados genérico.

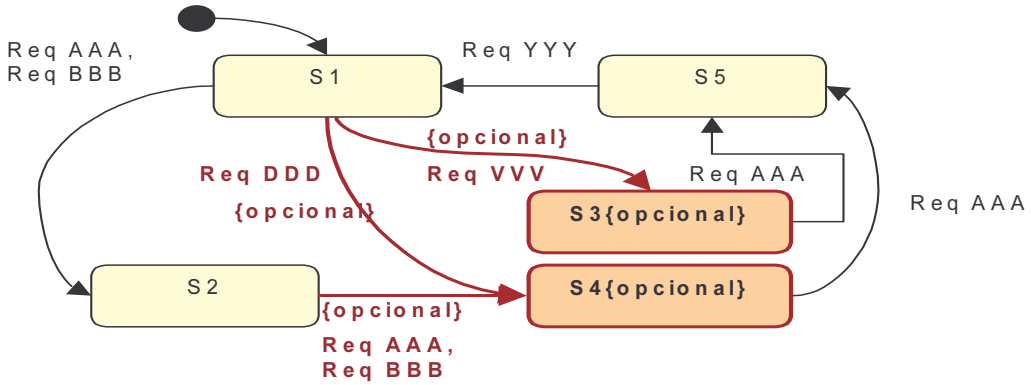


Figura 3.13 Diagrama de estados genérico con transiciones opcionales

El método de prueba utiliza los elementos básicos de la notación de los statecharts de UML, a saber estados, transiciones y eventos, y por ello, no se ha visto afectado por la nueva versión de UML 2.0 en sus aspectos fundamentales. El único aspecto donde hay discordancia con lo propuesto en UML 2 es en la representación de los diagramas de estados genéricos y específicos, que en UML se representan rodeados por una línea discontinua.

3.3.3.1 Guía de trabajo con statecharts

Se sugiere seguir las siguientes normas al utilizar los diagramas de estados jerárquicos o statecharts en el contexto del método, tomando en cuenta las directivas generales de [Dou99] (pp 357) y [Amb03] (pp. 82-91).

- Estados y Subestados

- Identificar el estado inicial.
- Colocar el estado inicial en la esquina superior izquierda del diagrama.
- Colocar el estado final en la esquina superior derecha del diagrama.
- Elegir para los estados nombres simples, pero descriptivos.
- Sospechar de los estados "pozo".
- Sospechar de los estados "milagro" (no se llega desde ningún otro estado).
- Utilizar subestados para reducir la complejidad.
- Incluir siempre estado inicial y final en el statechart superior de la jerarquía.
- Usar estados anidados si un mismo evento se aplica a varios subestados, o las actividades de entrada o salida de varios subestados coinciden.
- Usar estados AND u ortogonales si varios grupos de estados se pueden considerar independientes.
- Al heredar comportamiento de estados no se puede eliminar transiciones, ni borrar estados, ni añadir nuevos supraestados. De lo contrario se rompe el principio de Liskov, según el cual la clase que hereda tiene todo lo de la clase padre.
- Al heredar comportamiento con estados, se pueden añadir transiciones, añadir actividades en transiciones o estados, cambiar las actividades en las transiciones o los estados y usar operaciones polimórficas (redefinidas en el subestado).

- Transiciones y actividades

  - Usar actividades de entrada para acciones que se ejecutan siempre al entrar en el estado.
  - Usar actividades de salida para acciones que se ejecutan siempre al salir del estado.
  - Usar las actividades de transición para el resto de las acciones.
  - Nombrar las actividades que sean internas al sistema utilizando las convenciones del lenguaje de programación.
  - Nombrar las actividades que se repercutan sobre el exterior del sistema (los actores) utilizando el lenguaje natural.
  - Modelar una transición recursiva si se quiere salir y volver a entrar en el mismo estado.
  - Nombrar los eventos utilizando el pasado, porque son anteriores a la transición.
  - Etiquetar la transición en el estado origen.

3.3.3.2 Derivación de Casos de Prueba a partir de los Statecharts

Aunque este aspecto se va a discutir con más detalle en la parte de herramientas, una forma práctica de proceder para hacerse una idea del conjunto mínimo de los casos de prueba que hay que obtener de un statechart es "desdoblarlo", como muestra la Figura 3.14, de tal forma que se muestren individualmente cada una de las transiciones entre estados con los eventos que las producen y sus requisitos asociados:

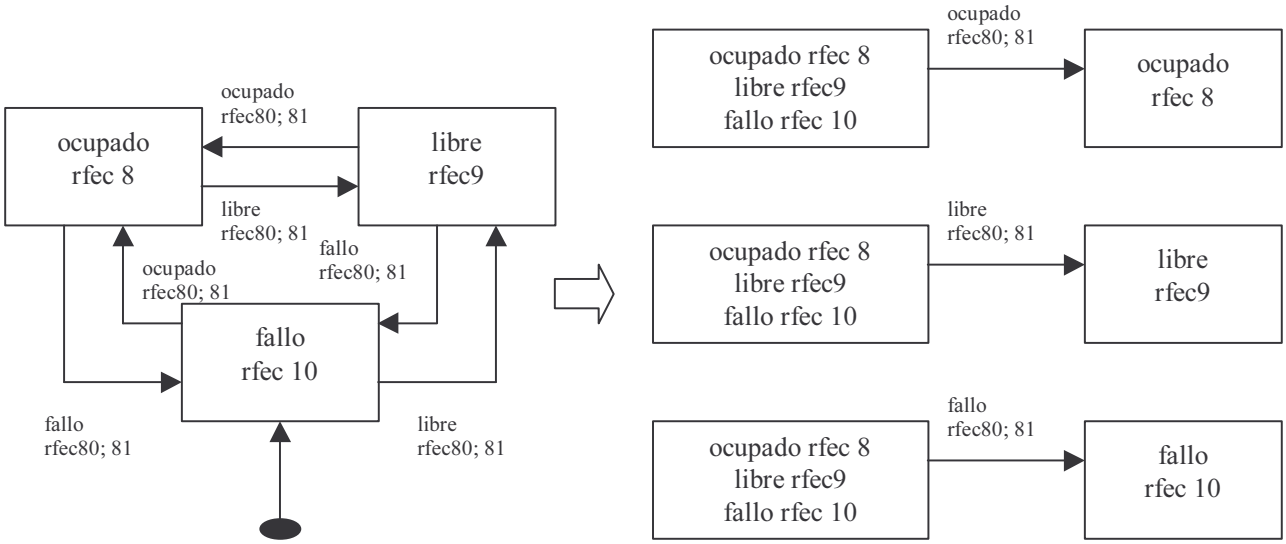


Figura 3.14 Obtención de casos de prueba a partir del diagrama de estados de la clase de prueba

3.3.3.3 Modelado de Requisitos de tiempo real mediante statecharts

Posibles requisitos de tiempo real son, por ejemplo:

- Tiempos de respuesta máximos del sistema ante determinadas entradas.
- Duración máxima de determinadas operaciones.
- Duración mínima de una condición de entrada para que el sistema la procese.

Los diagramas de estados jerárquicos se pueden utilizar para modelar estos requisitos de tiempo real, utilizando los eventos de tipo *timeout*, que son eventos del diagrama de estado que se producen pasado un tiempo o se repiten con un ciclo determinado.[HaPo98] modela los *timeouts* con statecharts usando la expresión *tm(E,T)*, donde E es un evento que se dispara cuando transcurre el tiempo T. Existe una actividad para deshabilitar los *timeouts*, *deltm*. Otra construcción propuesta en [HaPo98] es *schedule(G,T)*, que planifica el evento G en el tiempo para que se ejecute cíclicamente cada T unidades de tiempo.

Para visualizar en el tiempo como cambia el valor del estado según se producen los diferentes eventos [Dou99] utiliza los diagramas de temporización o "*timing diagrams",* de los cuales se recoge un ejemplo en la Figura 3.15, donde *Si (i=0,1,2,3,4)* son los estados y e_j *(j=0,1,2,3,4,5)* los eventos. Estos diagramas forman parte ya de UML 2 [OMG03].

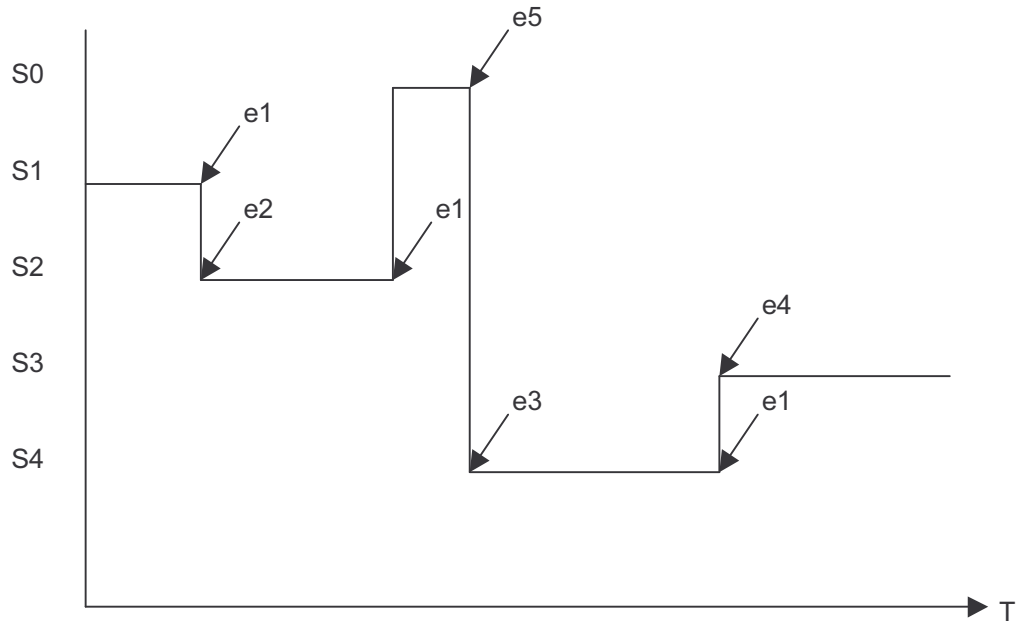


Figura 3.15 Diagrama de temporización

Al usar en las pruebas de requisitos de tiempo real un modelo de diagramas de estados es importante saber el grado de precisión con el que los tiempos de reacción de la implementación se ajustan a los especificados en el *statechart*. De no ser así, se estaría utilizando un modelo erróneo.

Algunos de los posibles factores de distorsión pueden ser, entre otros, el tiempo que tarda el sistema en cambiar de un estado a otro, y los tiempos de generación y envío de los eventos, que teóricamente son nulos [HaPo98]. Sin embargo, en sistemas de tiempo real donde hay una serie de tareas que se ejecutan cada ciclo en un orden fijo (ejecutivo cíclico), si se genera un evento para producir un cambio de estado, puede ocurrir, según como sea la implementación, que el evento se procese en el siguiente ciclo, lo que introduce un retardo en el cambio de estado, que será significativo o no según sea la duración del ciclo frente al tiempo requerido para la reacción del sistema.

3.3.4 Escenarios

Un escenario representa una de las posibles interacciones entre diferentes objetos del sistema. Por ejemplo, en un escenario podría verse cómo el objeto de la clase A manda un mensaje a través de un canal de comunicación a otro objeto clase B, que responde con un acuse de recibo. El problema que tienen los escenarios es que muestran sólo una de las posibles formas en que puede transcurrir el diálogo. Volviendo al ejemplo anterior, puede suceder que B espere el mensaje de A y que no le llegue, en ese caso, genera una excepción y declara el canal de comunicación como averiado, que es otro nuevo escenario. El número potencial de escenarios no está limitado a priori, y puede variar en función de los requisitos.

Se pueden clasificar los escenarios en:

- Estándar: El escenario sigue el flujo de ejecución normal. En el caso del ejemplo, sería cuando el mensaje del objeto A es recibido por el objeto B.
- Alternativos: El flujo de ejecución es diferente del escenario estándar, pero no se produce un error.
- Disruptivos: La ejecución normal se interrumpe debido a un error.

De esta forma, a la hora de probar los escenarios se puede acotar el número de pruebas poniendo como objetivo verificar una o varias de las clases de escenarios, por ejemplo, sólo los escenarios estándar, o los estándar y los alternativos, etc. Una vez que se han seleccionado los escenarios que se desea probar, la prueba consiste en verificar que la secuencia de envío de mensajes entre los objetos del escenario se ajusta a lo especificado en el modelo.

Con el fin de tener trazabilidad entre las pruebas y los requisitos, de la misma manera que en los diagramas de estados, se incluye en las interacciones del escenario información de requisitos, de tal forma que se consiga trazabilidad entre la prueba que se haga de la interacción y su requisito asociado.

Esta última técnica no se ha podido validar en la práctica, pues en el ejemplo industrial incluido en la Tesis Doctoral sólo se ha trabajado con diagramas de estados. Este aspecto, junto con una estrategia de selección de escenarios relevantes para las pruebas, forma parte de los posibles trabajos futuros continuación de la Tesis Doctoral.

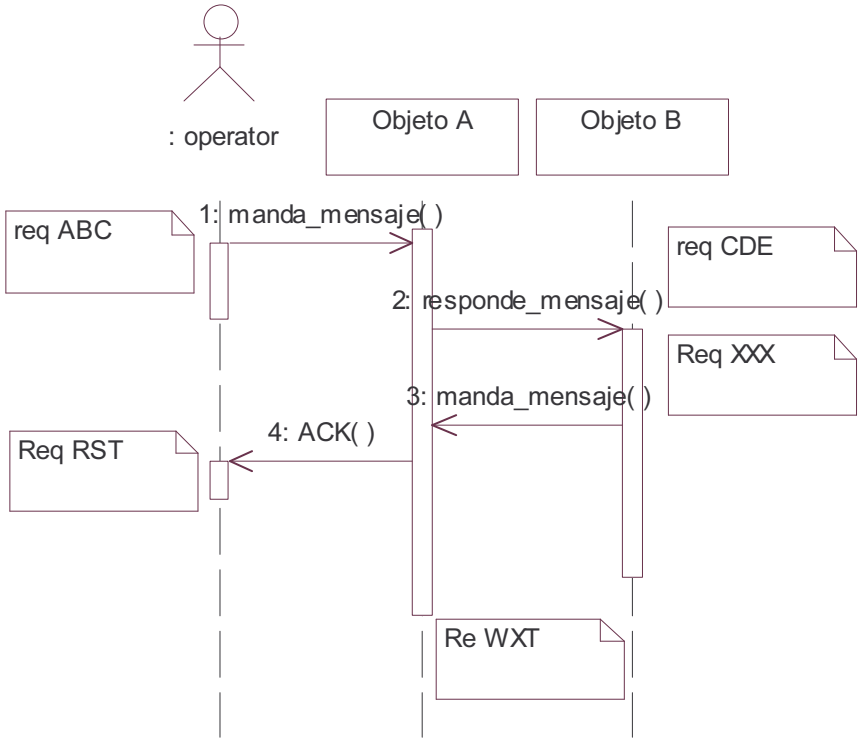


Figura 3.16 Escenario con información de requisitos

3.3.5 Reducción del conjunto de pruebas

Como se ha dicho ya, un conjunto de pruebas puede ser según [ITU97] exhaustivo (toda implementación que lo pasa es conforme), robusto (toda implementación que no lo pasa es no conforme) o completo (exhaustivo y robusto a la vez). El método de pruebas propuesto no busca conseguir conjuntos de pruebas de estas características, sino que busca optimizar en la práctica de la forma más estructurada posible el tiempo y los recursos disponibles para las pruebas. Para eso es preciso definir una estrategia de reducción del conjunto de pruebas a partir del conjunto de pruebas exhaustivo ideal.

[ITU97] propone las siguientes estrategias de reducción del conjunto de pruebas:

- Especificar explícitamente un modelo de fallos, es decir un subconjunto de implementaciones no conformes a partir de la especificación también llamadas *especificaciones mutantes*.
- Escoger un subconjunto aleatorio del conjunto de pruebas posible.
- Modificar la especificación de forma que imponga menos requisitos a la implementación y generar las pruebas a partir de especificación más permisiva.
- Permitir más variaciones en el comportamiento de la implementación.
- Hacer un postulado de pruebas o "*test assumption*" más "fuerte", lo que significa postular que la implementación se puede modelar mediante un número más limitado de formalismos.

De entre las estrategias escogidas por el método propuesto, la que se emplea es la de "fortalecer" la "*test assumption"* y la de especificación de un modelo de fallos. En primer lugar se elaboran los modelos de máquinas de estados y se asume que no hay más modelos de la implementación posibles, cosa que lógicamente no es cierta desde el punto de vista teórico, pero que nos limita el esfuerzo desde el punto de vista práctico. En segundo lugar, el modelo de fallos que se va a suponer es que tras un número finito de transiciones en la máquina de estados asociada a la clase de prueba, la implementación no esté en el estado correcto. Los detalles de este recorrido limitado de la máquina de estados se explican más adelante. Como toda opción de tipo pragmático, la estrategia de reducción de pruebas sólo se puede comprobar que funciona bien mediante la aplicación práctica del método a la Línea de Producto Software que se prueba.

Hay una propuesta en este sentido de reducción del conjunto de pruebas recogida en [CA03] consistente en definir una función f que establezca de acuerdo a un criterio determinado una prioridad de un conjunto de pruebas, de forma que dado un conjunto de pruebas T, se pueda encontrar un subconjunto T' tal que f(T`) > f(T). En principio, todos los casos de prueba generados tienen la misma prioridad, pero podrían establecerse prioridades, por ejemplo en función del tipo de requisitos. Desde el punto de vista de los sistemas de control del tráfico ferroviario, un posible esquema de prioridades se definiría estableciendo en los requisitos una distinción de niveles de seguridad, como la que propone la norma CENELEC, que establece un SIL de 0 a 4 (*Safety Integrity Level*). De esta forma se probaría con mayor prioridad los requisitos de SIL 4 (los más críticos). Otro posible criterio sería dar prioridad a las pruebas de los requisitos genéricos frente a los requisitos específicos, o viceversa.

Por ello, habida cuenta de sus limitaciones a nivel teórico, este método no se concibe como un sustituto sino como un complemento de las pruebas hechas de modo más intuitivo por parte de las personas responsables de las pruebas, que proceden en función de su experiencia y de las características concretas de la implementación bajo prueba, probando por ejemplo, lo que se ha cambiado en la implementación respecto de la versión anterior con especial meticulosidad.

3.3.6 Modelo de fallos

A la hora de reducir el conjunto de pruebas, se van a asumir los siguientes tipos de fallos en implementación, en referencia al modelo de diagrama de estados, dado el diagrama de estados jerárquico *D,* definido por un conjunto de transiciones *T* caracterizadas por su estado inicial *Ei*, evento *e* y su estado final *Ef*

$$D \subset E \; x \; M \; x \; E'$$

Siendo $E' = \{E_1, E_2,...,E_n\}$

Es decir, todos los estados posibles del diagrama de estados excepto el estado inicial, que no puede ser estado final por definición.

$D= \{T_0, T_1, T_2,...,T_j\}$ donde cada transición $T'= \{E_i,e_k,E_f\}$

siendo E el conjunto de todos los estados (considerando los subestados como un estado más) del diagrama de estados

$$E= \{E_0 , E_1, E_2,...,E_n\}$$

y M el conjunto de todos los eventos o mensajes válidos

$$M=\{e_0, e_1, e_2,...,e_m\}$$

y puesto que se está modelando el comportamiento de un sistema de tiempo real, dentro del conjunto de los eventos válidos existirán algunos de ellos asociados a condiciones temporales o plazos críticos

$$T \subseteq M = \{ t_0, t_1, t_2....\}$$

los fallos que se van a considerar son los siguientes:

- Un evento produce un estado final válido, pero que no es una transición de la máquina de estados

$$\exists\, t =\{A, a, B\} \mid (A \in E) \wedge (B \in E) \wedge (a \in M) \wedge (t \notin D)$$

- Un evento válido es ignorado

$$\neg \exists\, t =\{A, a, B\} \mid (A \in E) \wedge (B \in E) \wedge (a \in M) \wedge (t \in D)$$

- Un evento válido produce un estado final indefinido.

$$\exists\, t =\{A, a, B\} \mid (A \in E) \wedge (B \notin E) \wedge (a \in M) \wedge (t \notin D)$$

- Un evento erróneo es aceptado.

$$\exists\, t =\{A, a, B\} \mid (A \in E) \wedge (B \in E) \wedge (a \notin M) \wedge (t \notin D)$$

- Un evento erróneo produce un estado indefinido.

$$\exists\, t =\{A, a, B\} \mid (A \in E) \wedge (B \notin E) \wedge (a \notin M) \wedge (t \notin D)$$

3.3.7 Cobertura de fallos

La cobertura de fallos es la medida normalizada de la exhaustividad de un cierto conjunto de pruebas respecto del modelo de fallos elegido, lo que significa que expresa de forma cuantitativa la capacidad del conjunto de pruebas para detectar fallos. En el caso ideal de que el conjunto de pruebas sea exhaustivo, la cobertura es del cien por cien.

En el caso del método de pruebas de la Tesis Doctoral, los casos de prueba consisten en recorrer transiciones en la máquina de estados y la función de cobertura, para una máquina de estados determinada será por tanto

$$C = Número\ de\ Casos\ de\ Prueba\ /\ Número\ Total\ de\ Casos\ de\ Prueba$$

La fórmula de cobertura es válida para diferentes estrategias de recorrido de la máquina de estados, tanto si los casos de prueba consisten en transiciones simples entre estados o en transiciones compuestas que incluyen secuencias de recorrido de estados, que es una estrategia de recorrido más exhaustiva. Por ello hay que tener en cuenta que aun alcanzando una cobertura del cien por cien, según la estrategia de recorrido elegida es posible que haya fallos que no se descubran, ya que se prueba con menos exhaustividad.

3.3.8 Coste del conjunto de pruebas

Los costes producidos al generar, mantener, implementar y ejecutar un conjunto de pruebas se expresan mediante el coste del conjunto de pruebas. La medida de coste final será la de horas/persona, ya que es la más cómoda y habitual desde el punto de vista de la gestión del desarrollo, pues emplear, por ejemplo, el coste en dinero supone hacer una labor de contabilidad que está completamente fuera de lugar ya que supone calcular los costes fijos de la sede de la organización, los impuestos, los costes laborales, etc.

Para obtener la medida de coste final en horas persona, primero hay que conocer el número de casos de prueba totales y luego estimar el número medio de horas persona por caso de prueba diseñado, implementado y ejecutado, incluyendo el tiempo invertido en la puesta a punto del entorno de pruebas.

Por tanto la función de coste Q es la siguiente:

$$Q = Número\ de\ Casos\ de\ Prueba * \lambda$$

Siendo λ el coste medio en horas persona de cada caso de prueba. Cabe pensar que el coste medio de cada caso de prueba será más elevado si la estrategia de pruebas es más exhaustiva, pero podría también no ser así, si se cuenta con un soporte de herramientas software que automatice alguna de las fases de prueba. Por ello no se puede definir un valor general basándose únicamente en suposiciones de tipo teórico, con más razón aún si además se quiere incluir en el coste de las pruebas el esfuerzo realizado en el mantenimiento de la trazabilidad entre casos de prueba y requisitos.

El seguimiento o gestión de las pruebas está relacionado indirectamente con el coste, en este caso con el control del coste. Una forma práctica de seguimiento de las pruebas es controlar la evolución en el tiempo de los siguientes valores:

- número de pruebas planificadas (para una determinada versión de software).
- número de pruebas diseñadas.
- número de pruebas realizadas desglosado a su vez en número de pruebas pasadas y número de pruebas fallidas.

3.3.9 Arquitectura de Pruebas y Puntos de Control y Observación

La Figura 3.17 recoge el esquema abstracto de arquitectura de pruebas propuesto por [ITU97]. Se entiende por arquitectura de pruebas el entorno en el que se prueba la implementación bajo prueba. La arquitectura de pruebas describe como la implementación probada está integrada dentro de otros sistemas durante las pruebas y como interactúa con ellos y con el ejecutor de las pruebas.

El ejecutor de Pruebas es la implementación de un conjunto de pruebas, que ejecuta los casos de prueba y observa los resultados y que interactúa directamente con el entorno de prueba a través de los PCO (Puntos de Control y Observación) e indirectamente con la implementación probada (IUT o *Implementation Under Testing*). La función del ejecutor de pruebas va a consistir en seleccionar una

clase de prueba, inicializar el estado de un determinado objeto, enviarle un evento y observar la respuesta del sistema, que puede consistir en un cambio de estado o en un cambio de estado y en el envío de otro evento hacia el exterior.

El entorno de pruebas es el contexto en el que está integrada la implementación probada y que realiza el cometido de comunicar al ejecutor de pruebas con la implementación probada, asociando los eventos que suceden en la interfaz del PCO con eventos que suceden en la interfaz del IAP (Punto de Acceso a la Implementación). PCO e IAP pueden coincidir, cosa que en el método propuesto no es fundamental. El número de PCO e IAP depende de la implementación, si bien lo deseable desde el punto de vista práctico es que sea lo más reducido posible.

Según [ITU95], un PCO se caracteriza mediante los mensajes a nivel lógico y a nivel de protocolo de comunicaciones que tiene asociados de acuerdo a lo especificado en su conjunto de pruebas y que [ITU95] denomina las ASP (Primitivas de Servicio Abstractas o *Abstract Service Primitive*) y las PDU (Unidad de Datos de Protocolo o *Protocol Data Unit*).

En sistemas de tiempo real, es importante asegurar que el comportamiento temporal de la implementación no se modifica significativamente al monitorizarla a través de los PCO. Una posible solución es, como sugiere [Bin00], que el ejecutor de pruebas corra en una plataforma hardware distinta que la de la implementación probada. Tiene la ventaja de que el procesador donde se ejecuta la implementación probada se libera de la carga que supone el ejecutor, y la desventaja de que hay que implementar un mecanismo de comunicación entre los dos equipos hardware, que puede distorsionar más aún la respuesta del sistema. En definitiva, no hay una solución estándar para este problema, pues las circunstancias pueden variar mucho de una implementación a otra.

Con esta definición, se caracteriza el PCO con la siguiente interfaz (en seudocódigo C). Se ha introducido, tomando la idea de [Cou00] el tiempo como parámetro opcional en las llamadas para mandar y recibir eventos del sistema, con el objetivo de tener datos para poder verificar los requisitos de tiempo real.

```
/* Declaracion del tipo evento */

typedef external_event char[5];

/* Declaracion del tipo Timeout */

typedef struct Timeout {   Value: time;
                           Timer_is_cyclical: boolean;
};

/* Manda un evento a la maquina de estados indicando su origen*/

int  Send_external_event(int  Sender_ID,  external_event  &e,  int
Object_ID);

/* Recibe un evento de la máquina de estados, que le indica su
destinatario y opcionalmente el tiempo */

int Receive_external_event(external_event &e, int Object_ID, time
Time);

/* Manda un evento de Timeout a la maquina de estados indicando su
origen y opcionalmente el tiempo */
```

```
int Send_Timeout(int Sender_ID, Timeout_event &T, int Object_ID);

/* Recibe un evento de Timeout la máquina de estados, que le indica
su destinatario y opcionalmente el momento en que se produce */

int Receive_Timeout(Timeout_event &T, int Object_ID, time Time);

/* Lee al estado del objeto */

State Get_State (int Object_ID);

/* Modifica el estado del objeto */

int Set_State (int Object_ID, State new_State);
```

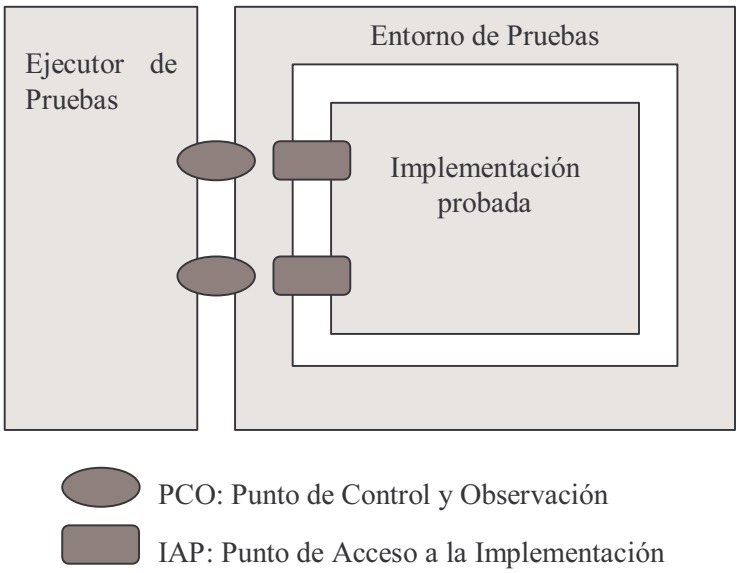


Figura 3.17 Arquitectura de Pruebas

El hecho de que la implementación probada pertenezca a una línea de Producto Software, influye en la arquitectura de pruebas. Se asume que se puede usar la misma arquitectura de pruebas para toda la Línea de Producto Software y lo deseable es que se puedan usar los mismos PCO e IAP para todos los productos. Si no fuera posible, se tendrían PCO genéricos o comunes y PCO e IAP particulares de cada producto, como muestra la Figura 3.18.

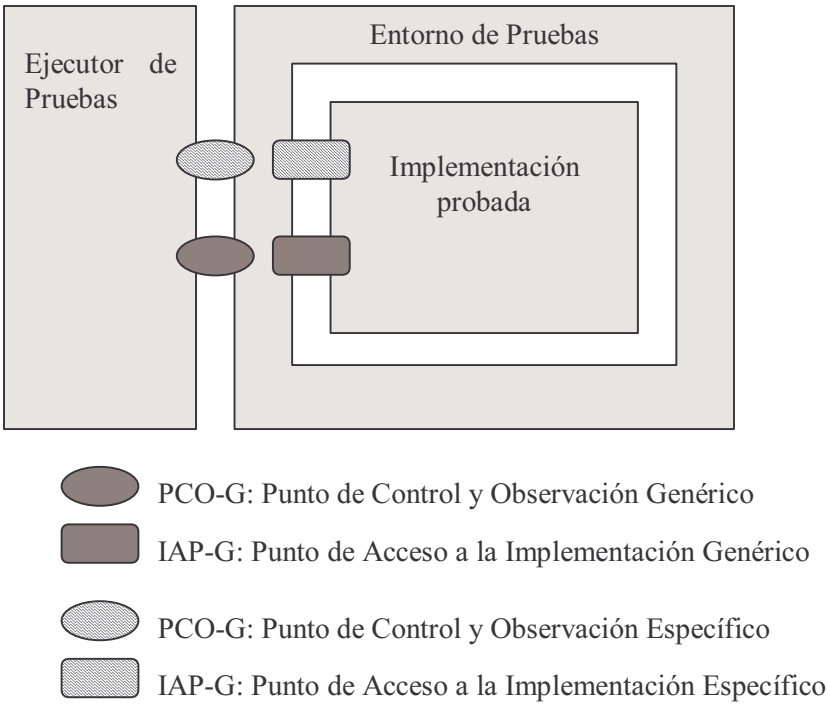


Figura 3.18 Arquitectura de Pruebas extendida para Líneas de Producto Software

3.3.10 Relaciones entre los elementos del método

La Figura 3.19 muestra las principales entidades del método de prueba y las relaciones que existen entre ellas. Los requisitos son el punto de partida de las pruebas y como estamos hablando de una Línea de Producto Software, pueden ser genéricos o específicos. Los requisitos están asociados con una determinada *clase de pruebas*, que modela el comportamiento del sistema mediante un *diagrama de transición de estados* y opcionalmente mediante uno o más *escenarios,* en los que pueden haber otros objetos *participantes* pertenecientes a otra *clase de prueba* y que por tanto se comportan según otro diagrama de estados distinto. A su vez cada escenario contiene uno o varios *casos de prueba*.

Partiendo del modelo de comportamiento se elaboran los *casos de prueba,* cuya implementación son los *datos de prueba* y que incluyen información de trazabilidad de requisitos, para evaluar la cobertura de las pruebas (cuántos requisitos genéricos y/o específicos se han probado, etc.). Los casos de prueba de escenarios incluyen como información específica la interacción objeto del caso de prueba, es decir, el objeto participante emisor, el objeto participante receptor y el evento o mensaje que se mandan. Los casos de prueba tienen asociado un *punto de control y observación* (PCO), que también puede ser genérico o específico (como se ha dicho ya, lo recomendable es que sea igual para todos los productos).

La clase recorrido *Diagrama de Estados* y *Recorrido Escenario* son agregaciones compuestas de las respectivas secuencias de casos de prueba que son necesarias para recorrer la máquina de estados o el escenario. Estas secuencias de recorrido de la máquina de estados o del escenario se determinan de acuerdo a una determinada *Estrategia* (todas las transiciones, todos los estados, etc.)

Siguiendo la línea de [DM04], donde se muestra la versión de este modelo publicada en [MeDu01], se ha añadido además las variaciones en los diagramas de estados de los diferentes productos, que

pueden suponer variaciones en los eventos, variaciones en los estados y variaciones en las transiciones.

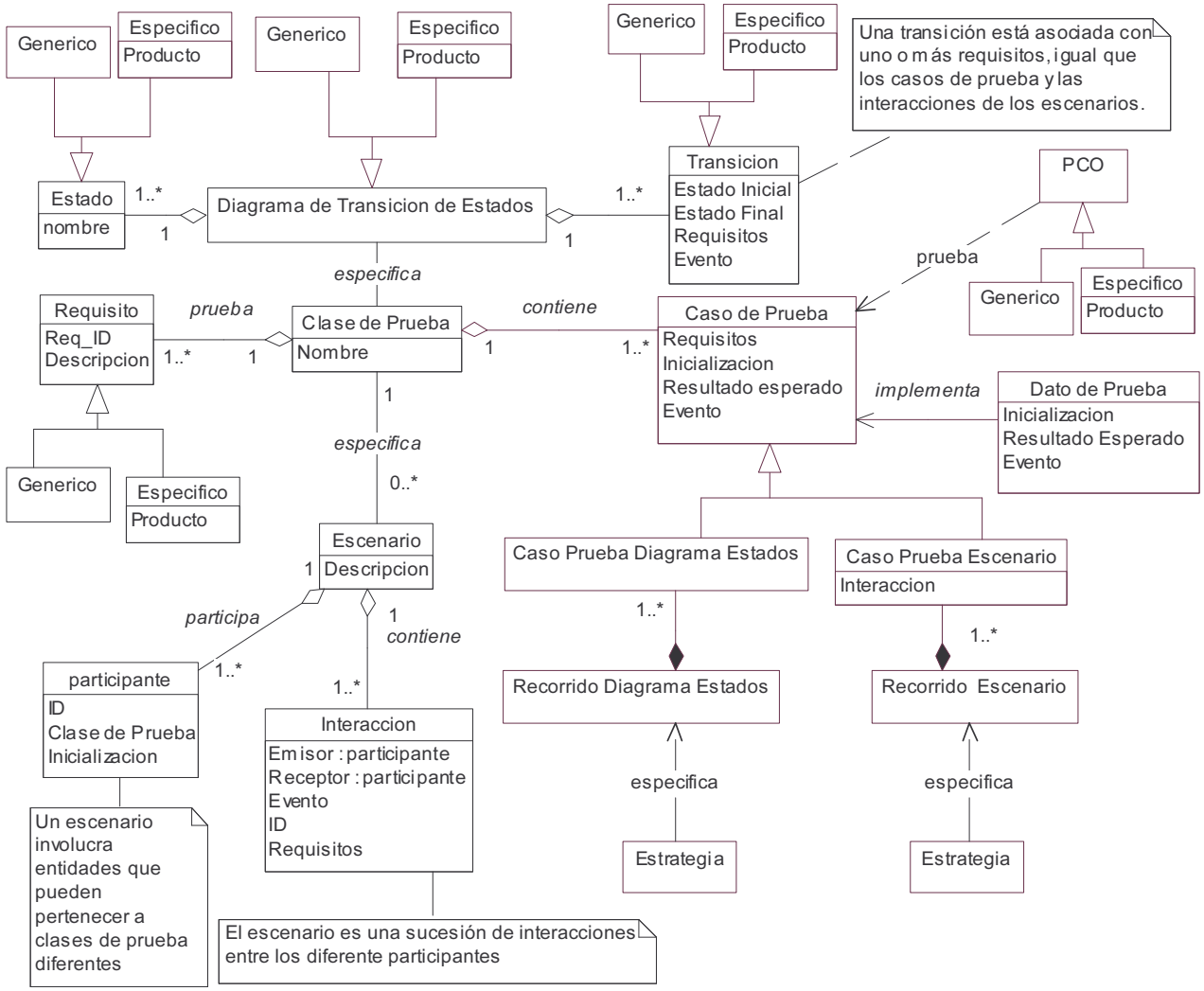


Figura 3.19 Metamodelo de artefactos de prueba y sus relaciones

3.4 Mejora del proceso software mediante la aplicación del método

En la década de los 90 el Software Engineering Institute (SEI) publicó el Capability Maturity Model for Software (CMM) [SEI91] para ayudar a las organizaciones a mejorar sus procesos de desarrollo de software. Este modelo describe un camino de mejoramiento evolutivo para pasar desde un proceso software caótico a un proceso disciplinado. El CMM proporciona un marco de evolución de la organización en cinco niveles de madurez que se describen en la siguiente tabla:

Nivel	Objetivo	Areas Clave de Proceso
5. Optimizado	Mejora Continua del Proceso	Prevención de Defectos Gestión de Cambios de Tecnología Gestión de Cambios de Proceso
4. Administrado	Proceso Predecible	Gestión de Cuantitativa de Procesos Gestión de Calidad Software
3. Definido	Proceso estándar y Consistente	Organización orientada a Procesos Definición de Procesos de la Organización Programa de Formación Gestión Integrada del Software Ingeniería de Producto Software Coordinación Intergrupos Revisiones
2. Repetible	Proceso Disciplinado	Gestión de Requisitos Planificación de Proyecto Software Seguimiento y Predicción de Proyecto Software Gestión de Subcontratas Software Aseguramiento de Calidad Software Gestión de Configuración Software
1. Inicial	Ad hoc, caótico	-

Tabla 3.4 Niveles CMM

CMMI es la versión de CMM aparecida recientemente [SEI03]. Se dividen las áreas de proceso de la organización en cuatro grupos, gestión de procesos, gestión de proyectos, ingeniería y soporte, y se evalúa cada área de proceso por separado, de forma que se puede concentrar el esfuerzo en una u otra área según las necesidades de la organización.

En [CA03] se recoge una experiencia industrial de validación de línea de producto software por la compañía Nokia aplicando el proceso de mejora del software CMMI [SEI03], si bien sin dar detalles concretos, definiendo como áreas de mejora del proceso software de Línea de Producto Software la validación y la verificación. La Figura 3.20 muestra las áreas de mejora junto al resto de las actividades del proceso.

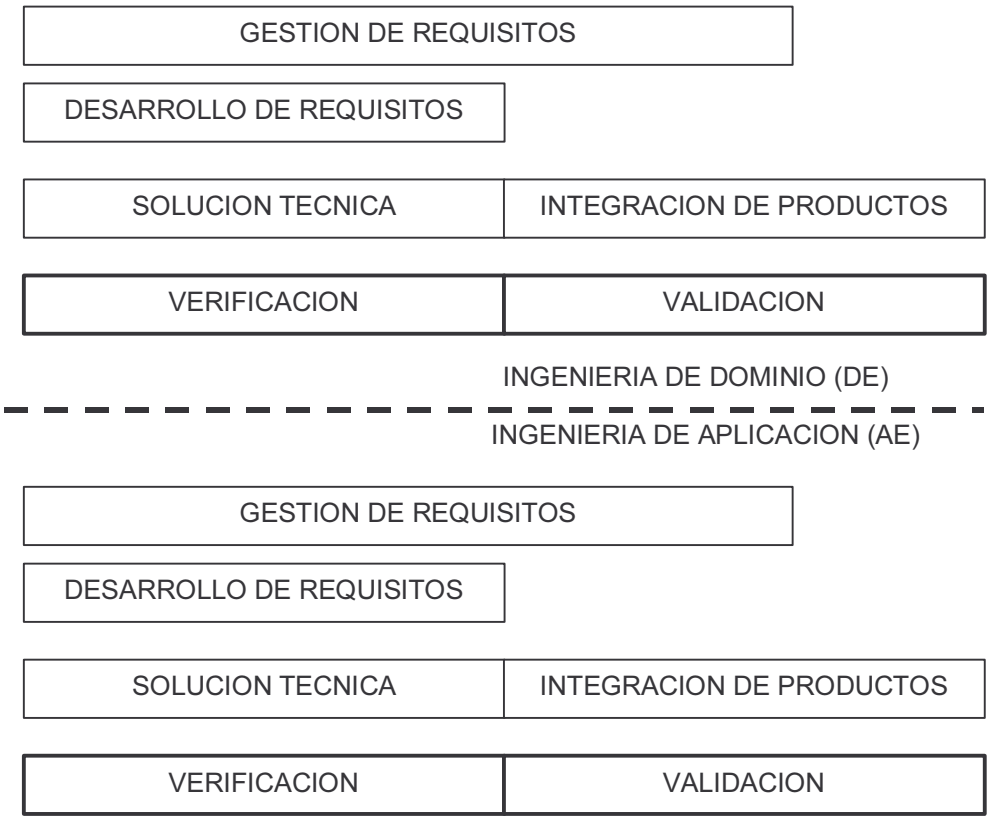


Figura 3.20 Areas de Proceso Software de Líneas de Producto

El método de Pruebas objeto de la Tesis Doctoral se puede incluir dentro del área de mejora de proceso software de validación, ya que propone un flujo de actividades para llevar a cabo la validación de Líneas de Producto Software. Este flujo consta de las fases de diseño de pruebas, implementación de pruebas y ejecución de pruebas. Las actividades relacionadas con el método son especialmente relevantes para los Niveles CMM 2 (Gestión de Requisitos, Seguimiento y Predicción del Proyecto Software) y 3 (Ingeniería de Producto Software).

3.5 Flujo de actividades del método de Pruebas

La Figura 3.21 muestra el flujo de actividades en las pruebas de sistema (validación), desde el diseño de las pruebas hasta su realización y es aplicable tanto a productos desarrollados individualmente como a Líneas de Producto Software. En el proceso hay tres fases diferenciadas, que son:

a) Diseño de Pruebas: a partir de la especificación de Requisitos, expertos del dominio elaboran modelos susceptibles de ser probados de forma sistemática como diagramas de transición de estados y escenarios.

b) Implementación de Pruebas: para cada caso de prueba se crea su correspondiente dato de prueba, conteniendo la información de la especificación del caso de prueba escrita en forma de tabla (usando, por ejemplo una hoja de cálculo, como [Buw99]). La diferencia entre caso de prueba y dato de prueba es que el caso de prueba describe la prueba desde un punto de vista funcional. El dato de prueba sin embargo, implementa la funcionalidad comprobada por el caso de prueba mediante algún servicio ofrecido por la implementación del sistema.

La Figura 3.21 incluye también un posible camino de generación de pruebas de forma no automatizada, que puede ser necesario si el entorno de automatización no está todavía maduro y se quiere tener una seguridad de que el entorno no está proporcionado resultados de prueba equivocados. Otro posible motivo para tener pruebas realizadas manualmente por una persona experimentada en el sistema puede ser una nueva funcionalidad cuya prueba no se haya podido automatizar por ser especialmente compleja, o por falta de tiempo.

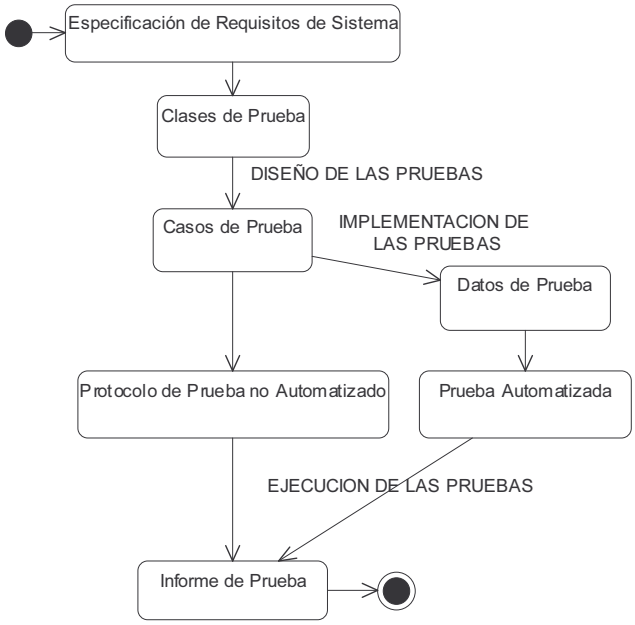


Figura 3.21 Actividades a partir del diseño de las pruebas hasta su realización

c) Ejecución de las Pruebas: Los scripts de prueba generados automáticamente a partir de los datos de prueba se ejecutan de forma combinada con la aplicación que se está probando. Los resultados se guardan en un fichero. La automatización de las pruebas, si existe, precisa de dos herramientas, el generador de casos de prueba y el analizador de resultados.

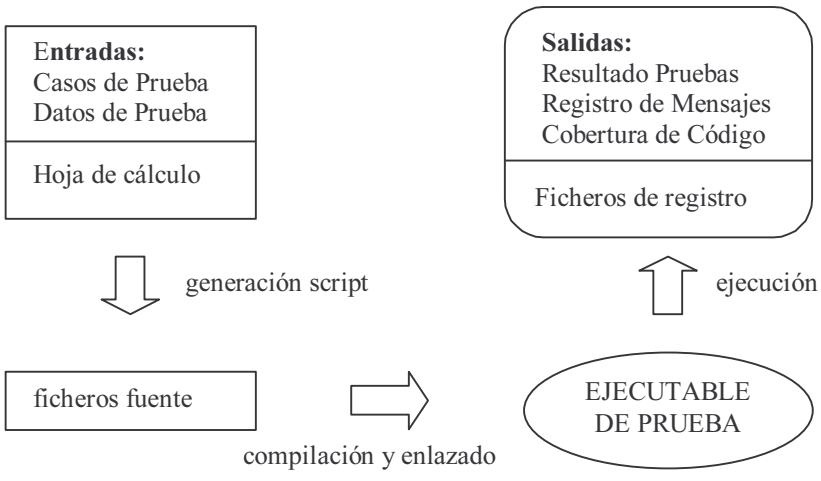


Figura 3.22 Esquema del Proceso de Automatización

El generador de casos de prueba transforma la especificación del caso de prueba que los validadores del sistema han expresado en forma de diagrama de estados en el dato de prueba, o lo que es lo mismo, en un conjunto de llamadas software procesable por la aplicación que se está probando. El analizador de resultados es otra de las herramientas que se propone para automatizar las pruebas. Se asume que el sistema que se está probando es capaz de registrar de alguna forma (un fichero de traza, etc.) la ejecución de los casos de prueba, y, que por lo tanto, va a ser posible comprobar si los casos de prueba que se han hecho han pasado o han fallado. Para un cierto dato de prueba *DP*, el analizador compara el estado final con el resultado esperado.

$$CasoPruebaPasa = compara(Lee\_Estado(Ef), Ef\_esperado)$$

Esta herramienta es de gran utilidad para controlar el avance de las pruebas: número de casos de prueba ejecutados, pasados y fallados. En la parte de objetivos prácticos de la Tesis Doctoral se da más información sobre estas herramientas.

Los casos de prueba así formulados pueden ser reutilizados en los distintos componentes de una familia de productos directamente si la clase de pruebas tiene el mismo comportamiento en cada uno de los productos. Tal vez sea necesaria una pequeña adaptación de los datos de prueba, en el caso de que los identificadores de los objetos no coincidan.

En el caso de que la clase de pruebas tenga un comportamiento diferente en los distintos productos, sólo se puede reutilizar el entorno de generación de casos de prueba y el analizador de resultados; puesto que las especificaciones de los casos de prueba serán dispares.

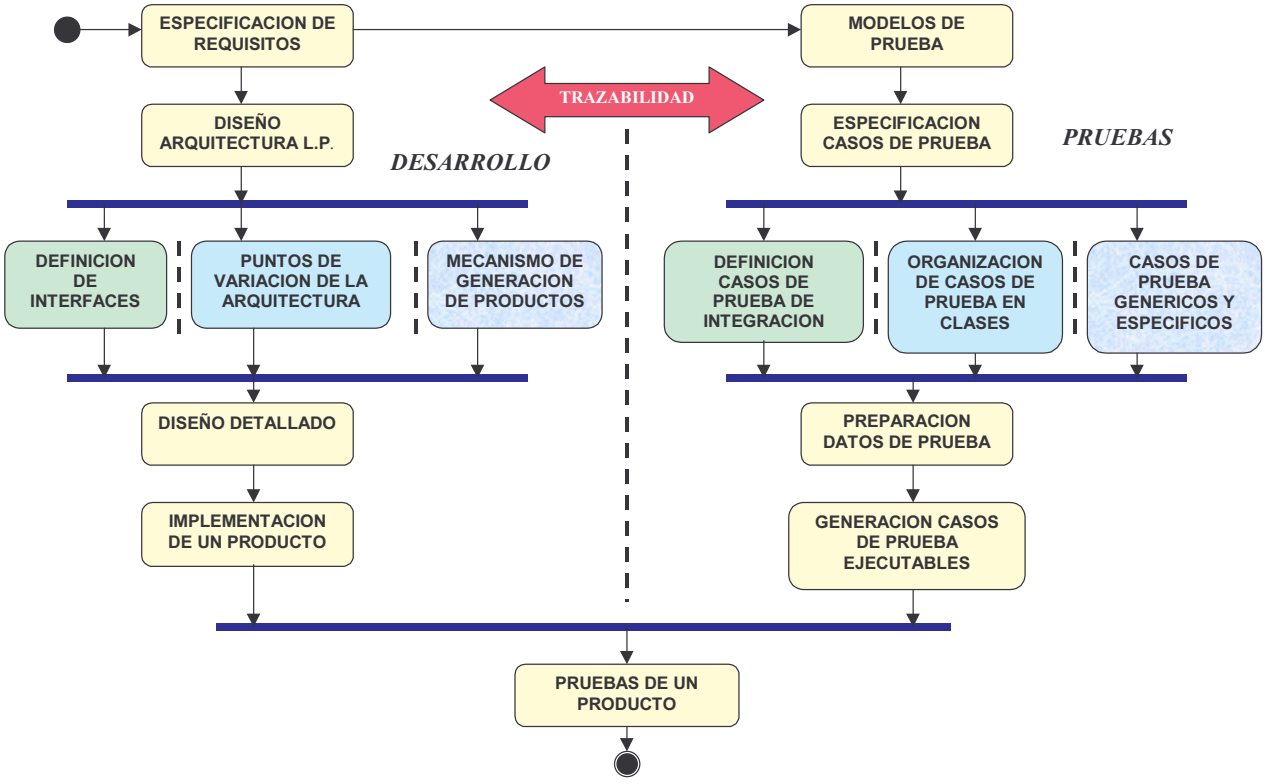


Figura 3.23 Actividades de desarrollo y prueba en una línea de productos software

La Figura 3.23 detalla el flujo de actividades de la Figura 3.21, en la forma de un diagrama de actividad de UML, los diferentes pasos que se dan a lo largo de las pruebas de una línea de productos

software. El proceso arranca con la especificación de los requisitos. A partir de ahí, se produce un desdoblamiento y las pruebas aparecen como proceso paralelo al desarrollo. El principio de la secuencia es la especificación de requisitos de toda la familia de productos y como final, para simplificar, se han fijado las pruebas de un producto individual. La línea discontinua del centro representa que las pruebas son en este método una actividad "paralela" al desarrollo, porque se entiende que este es el enfoque más efectivo. Las actividades de desarrollo que aparecen en la Figura 3.23 son las que se han considerado especialmente relevantes para las pruebas.

Este esquema no podría aplicarse en sentido estricto en algunas circunstancias (como por ejemplo, si el equipo de desarrollo tiene un tamaño muy reducido y no existe un equipo de pruebas independiente), pero seguiría siendo válida la idea de un diseño de las pruebas fuertemente interrelacionado, trazable, con el proceso de desarrollo. La flecha horizontal entre el desarrollo y las pruebas simboliza la trazabilidad existente entre los requisitos y los casos de prueba. Al final, ambos procesos convergen cuando se realizan las pruebas al producto final.

La especificación de requisitos es el paso previo al comienzo del diseño de arquitectura y a la elaboración de los modelos de prueba en forma de máquinas de estados y escenarios. En el método que se propone se utilizan estas dos técnicas semiformales, pues se consideran efectivas y que gozan de amplia aceptación en el ámbito académico y en el industrial. De la misma forma que diferentes analistas pueden diseñar arquitecturas diferentes a partir de unos mismos requisitos, los modelos de prueba pueden ser muy variados. Es un problema que no tiene solución única. Ahora bien, los modelos aunque puedan tener variaciones desde el punto de vista formal, deben de ser equivalentes desde el punto de vista semántico: en caso contrario, los requisitos serían ambiguos o contradictorios.

La estructura de los casos de prueba ha de reflejar la de los requisitos. Si se define una jerarquía de requisitos genéricos y específicos, los casos de prueba tienen la misma estructura. La estructura de los requisitos sirve para organizar los casos de prueba, pues define clases funcionales que van a compartir un mismo modelo de prueba (un diagrama de estados o un conjunto de escenarios).

En la fase de diseño de arquitectura se define la arquitectura genérica de la línea de productos y se establecen los mecanismos para obtener los distintos componentes de la familia de productos a partir de la arquitectura común. Este diseño de la arquitectura sirve como elemento de referencia en la organización de los casos de prueba en clases de prueba (funcionales, de rendimiento, etc.). Dentro de esta fase de diseño de arquitectura destacan las tres actividades siguientes:

- Definición de las interfaces: Durante la fase de pruebas, especialmente si se va a automatizar, se va a acceder a las interfaces del sistema para enviar y recibir mensajes, para monitorizar los mensajes que el sistema intercambia con el exterior. Se pueden mandar mensajes intencionadamente incorrectos para comprobar la robustez del sistema.

- Identificación de los puntos de variación: Son aquellas partes de la arquitectura que son diferentes de un producto a otro. Hay diferentes mecanismos para implementar los puntos de variación. Posibles técnicas son la inserción de un componente en lugar de otro, herencia, parametrización, etc.

- Mecanismo de generación de productos: Las pruebas deben de verificar que este proceso está exento de errores. Puede ser una ayuda que la implementación ofrezca servicios de identificación de la versión de software.

La especificación de las interfaces internas y externa es la base de la definición de casos de prueba de integración. Si las interfaces tienen variaciones de unos productos a otros, habrá que tenerlo en cuenta a la hora de definir los casos de prueba de integración. Para los casos de prueba de integración (prueba de las interfaces) se propone como primera solución un enfoque convencional de pruebas usando un protocolo escrito, para verificar que existe comunicación a través de la interfaz. Comprobar que,

además de existir comunicación a través de las interfaces, el comportamiento del sistema es correcto, se hace mediante los casos de prueba automatizables que se describen en el apartado anterior.

Paralelamente (no en sentido estricto, puede hacerse también a posteriori) al diseño de la arquitectura, se diseña la arquitectura del "banco de prueba" software o "testware". Este término denomina a todo aquel software que se implementa para automatizar las prueba y que simula de alguna manera el entorno exterior al sistema que se está probando. Muchas veces será tanto o más complejo que el sistema que se prueba. Un conocimiento exhaustivo de la arquitectura permitirá integrar el "banco de prueba software" más fácilmente.

Las variaciones de la arquitectura y el mecanismo de obtención de los distintos miembros de la familia en la fase de desarrollo se corresponden en la fase de pruebas con la asignación de casos de prueba a cada uno de los componentes de la línea de productos.

El diseño detallado en la fase de desarrollo se corresponde, como indica la Figura 3.23 con la generación de datos de prueba. Para automatizar las pruebas, debe de poderse tener acceso a la información del diagrama de la Figura 3.23, en la que aparecen todos los elementos del método que se propone. Al final, todos los casos de prueba diseñados se recogen en un protocolo y se ejecutan sobre un elemento de la familia de productos ya implementado.

El trabajo práctico de la Tesis Doctoral está centrado en el proceso de la Figura 3.21 y está dentro de la zona marcada por la línea de puntos en la siguiente figura. Se va a proponer un conjunto de herramientas que permiten la confección de los casos de prueba a partir de las especificaciones elaboradas en forma de máquinas de estados, y que a su vez, manejan la información de los requisitos acerca de las características comunes y específicas de cada producto.

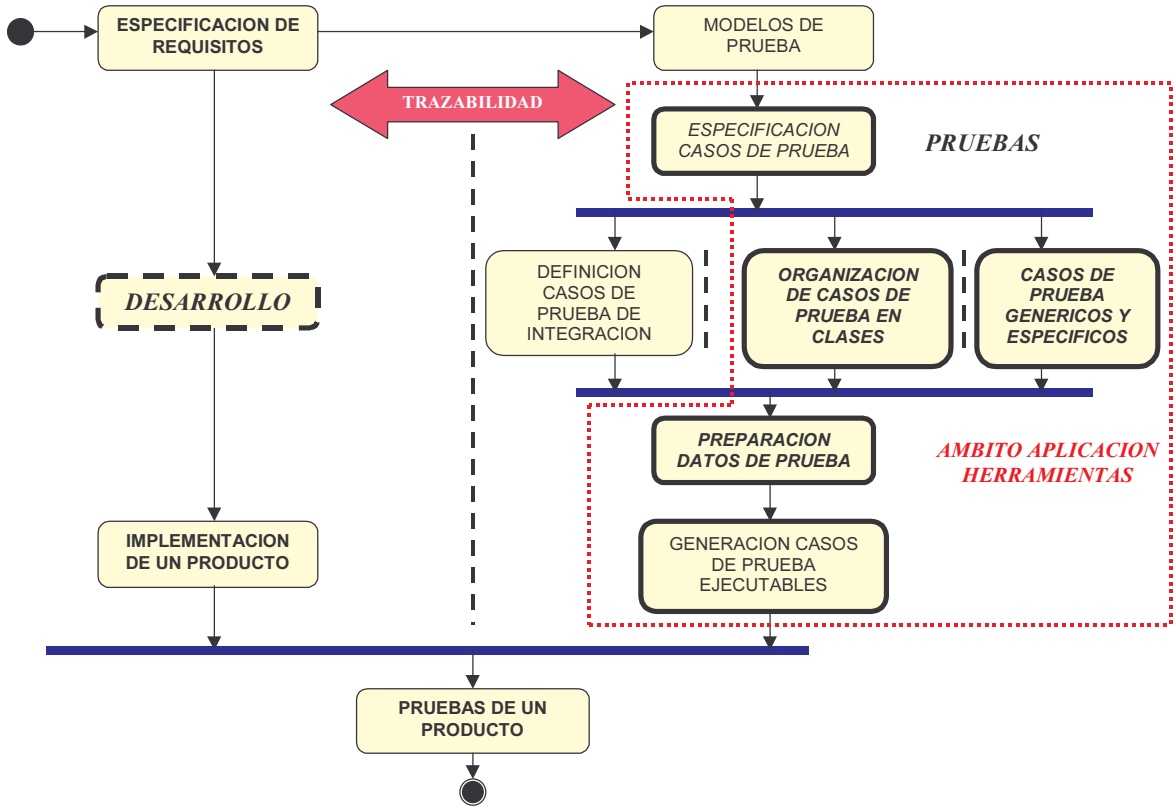


Figura 3.24 Area de trabajo práctico preferente

A modo de resumen:

- Es muy interesante probar las Líneas de Producto Software de forma automática, pues los beneficios de la automatización de pruebas software se tienen no en un sólo sistema, sino en varios.
- Las pruebas de Líneas de Producto de sistemas software pueden modelarse con diagramas de estados.
- Los diagramas de estados pueden ser statecharts (diagramas de estados jerárquicos de UML), incorporando mayor capacidad de representación.
- Es posible probar los modelos basados en statecharts de forma automática.
- Si se modela la línea de productos con statecharts, especificando con esta técnica sus requisitos funcionales de comportamiento, es posible validar la línea de productos software de forma automática.
- La trazabilidad de requisitos y casos de prueba, junto a una organización de requisitos que permita diferenciar fácilmente que requisitos son generales y cuáles son específicos, sirve para medir la cobertura de requisitos de las pruebas de cada producto. Esto es útil para medir el progreso de las pruebas y para posibles procesos de certificación y validación en los que dicha cobertura de requisitos se deba acreditar.

4 Aplicación del Método de Pruebas en un Caso Industrial

4.1 Introducción

El dominio de aplicación elegido es el de los sistemas software de control del tráfico ferroviario, en el que el autor de esta Tesis Doctoral lleva desarrollando su actividad profesional en distintos campos de la ingeniería del software (desarrollo, pruebas, aseguramiento de calidad, especificación de requisitos, colaboración en proyectos de carácter internacional) desde el año 1993.

Los ferrocarriles y los sistemas ferroviarios de transporte colectivo están proporcionando una importante movilidad en un mundo de atascos. Integrando de forma inteligente tecnologías avanzadas de señalización y de telecomunicación, las administraciones ferroviarias demandan productos y servicios que aumentan considerablemente la capacidad de tráfico a la vez que reducen los costes de infraestructura.

El tráfico de viajeros y de mercancías se ha duplicado en los últimos 20 años y aún sigue creciendo a gran velocidad. Para generar beneficios en un mundo privatizado y desregulado, los operadores demandan soluciones optimizadas para el transporte ferroviario, ya sea de cercanías, metros, líneas regionales o líneas de larga distancia – en especial las líneas de alta velocidad -, mientras se preparan para la normalización y la interoperabilidad para promover la competencia. Entre tanto, el desarrollo tecnológico, basado en su gran mayoría en la tecnología de la información, está revolucionando el modo de funcionamiento de los sistemas ferroviarios.

Este cambio tan rápido depende, en gran parte, de las soluciones avanzadas de señalización ferroviaria. Las soluciones que no necesitan cables gozan cada vez de mayor aceptación, y las aplicaciones telemáticas se utilizan para la supervisión a distancia, la detección de averías y el mando centralizado. Cada vez se instalan más dispositivos inteligentes a bordo del tren que en la vía, con objeto de conseguir mayor precisión y menores costes de mantenimiento.

Los análisis del mercado apuntan hacia un futuro con una demanda de servicios que exige el paso a una nueva generación de productos. Esta generación ha de incorporar las últimas tecnologías en el terreno de la electrónica, las comunicaciones y el software que sean capaces de satisfacer los requisitos de seguridad que demandan los sistemas de señalización ferroviaria y que a la vez sean más competitivas que las existentes en coste y prestaciones.

Otra de las grandes demandas del mercado ferroviario en los próximos años es la interoperabilidad entre los distintos sistemas de señalización de los diferentes países, especialmente los de alta velocidad, fuertemente impulsada por la unión europea con los estándares ERTMS (European Railway Traffic Management System) y ETCS (European Train Control System).[EC96][UN00]

En los sistemas de control de tráfico ferroviario van a seguir vigentes las exigencias de los estándares de seguridad y calidad en todo el proceso de desarrollo. Los principales son:

- Normas Europeas CENELEC, que contiene prescripciones respecto a las diferentes actividades que se llevan a cabo a lo largo de la trayectoria del producto. Existen normas CENELEC específicas para software de sistemas ferroviarios que las empresas activas en este sector están implantando (prEN 50126 y 50128).

- Norma ISO 9000, que regula las características del sistema de gestión de la calidad de una organización, y que es una exigencia habitual por parte de los clientes (administraciones ferroviarias) a las empresas suministradoras.

En esta parte de la Tesis Doctoral se describe el dominio de aplicación de control del tráfico ferroviario en general y cómo es posible aplicar en este dominio las técnicas propias de las Líneas de Productos software. Posteriormente se explica como se ha aplicado el método propuesto en la Tesis Doctoral con el fin de demostrar su validez.

4.1.1 El control del tráfico ferroviario

El ferrocarril se caracteriza por el uso de las vías como sistema de guiado de los vehículos y por las largas distancias de frenado a causa del débil rozamiento que existe entre las ruedas de los trenes y los carriles de las vías férreas. Para una velocidad de 160 km/h la distancia de frenado es del orden de 1000 metros. El tren es especialmente adecuado para transportar grandes volúmenes de personas o mercancías de forma rápida.

A diferencia del automóvil, el tren sólo puede ir hacia adelante y hacia detrás. El cambio de un carril a otro se hace por medio de las agujas, unos dispositivos mecánicos situados en las bifurcaciones de la vía férrea que, según como estén situadas, desvían el tren en una dirección o en otra. La ruta del tren no la fija el maquinista, sino la persona encargada de controlar el tráfico ferroviario, normalmente con ayuda de uno o varios sistemas automáticos para garantizar la seguridad y optimizar la explotación de la red. El maquinista recibe información del estado del trayecto ferroviario por medio de señales de diferentes tipos (ópticas y acústicas) según la cual puede acelerar o frenar el tren.

La red ferroviaria esta constituida por el conjunto de todas las vías y las agujas que las conectan. Se forman así nodos de conexión en la red ferroviaria que son las estaciones, de las que hay muchos tipos en función del tamaño y del uso que se les da. Las vías que unen unas estaciones con otras se denominan vías de bloqueo o secciones de bloqueo. Donde la vía se cruza con una carretera por la que pasan coches o personas, es habitual que haya un paso a nivel, que puede ser con o sin barreras. Los dispositivos que el tren encuentra a su paso que son relevantes para que el tren fije su velocidad y su dirección se llaman elementos de campo (p. ej. señales, agujas, pasos a nivel, etc.).

Las largas distancias de frenado de los trenes sobrepasan habitualmente el trayecto que ve el maquinista. Por ello, un aspecto esencial en el tráfico ferroviario es el asegurar que los trenes guardan entre sí una distancia suficiente. Cuatro maneras de conseguir esto son:

a) Marcha a la vista

Es el método más antiguo. El conductor del tren que va por detrás regula manualmente la distancia que guarda respecto del tren que circula por delante. Este método se usa en los tranvías y en trenes que circulan a poca velocidad (entre 25 y 40 km/h) y también maniobrando dentro de una estación.

b) Separación temporal

Es un método que surgió también en los albores del ferrocarril, cuando la ocupación de las vías era aún muy débil y no existían otras posibilidades para regular la separación de los trenes. Consiste en que cada tren espera un tiempo mínimo después de que haya salido el tren precedente. En Europa este método no se usa desde el siglo 19, pero en Norteamérica se emplea con ciertas restricciones en el tráfico de mercancías.

c) Separación por distancia fija

Consiste en que dos trenes consecutivos guardan entre sí como mínimo la distancia máxima de frenado, a la que se le añade un margen de seguridad. Para poder conseguir esto, es necesario contar con sistemas que supervisen continuamente de forma segura la posición de los trenes. La adopción de tecnologías de ubicación basadas en transmisión por radio va a extender este método en el futuro.

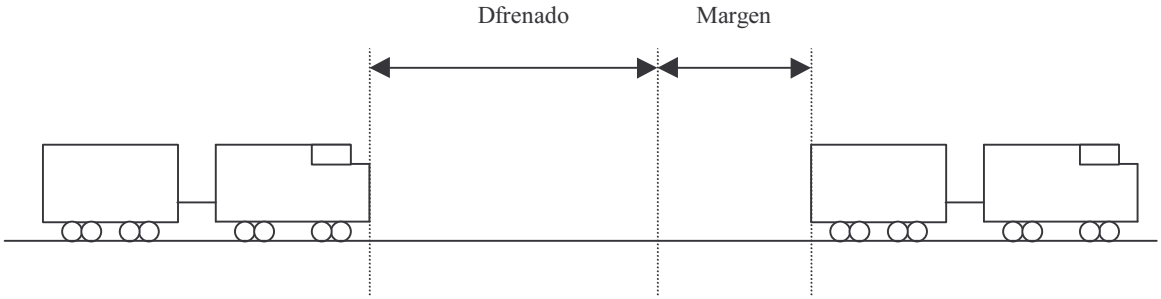


Figura 4.1 Separación por distancia fija

d) Separación por secciones de bloqueo

La separación entre trenes se garantiza por medio de señales (lo que el profano llama semáforos) fijas situadas a lo largo del trayecto que dividen la vía en secciones que tienen dicha longitud y en las que no puede haber nunca más de un tren. Estas señales se denominan señales principales o señales de bloqueo. La distancia entre dos trenes consecutivos es igual a la distancia de frenado aumentada por el margen de seguridad, a la que se añade la longitud de la sección de bloqueo. Este es el método más empleado actualmente. Para aumentar el número de circulaciones, se puede colocar en las secciones de vía señales intermedias de forma que sea posible tener más de un tren en cada sección.

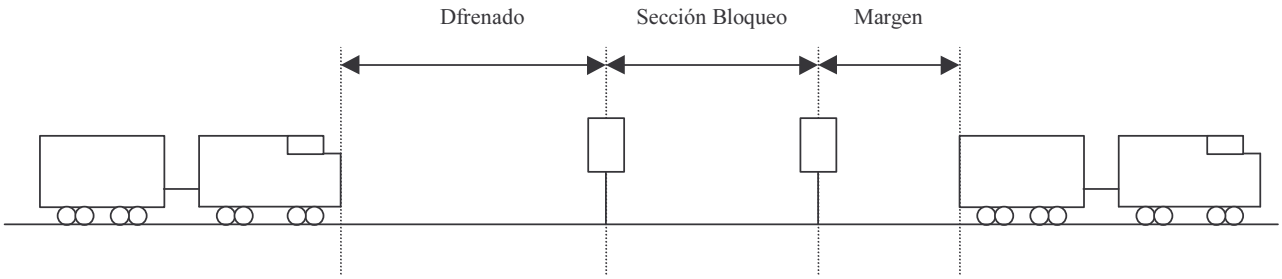


Figura 4.2 Separación por secciones de bloqueo

El control del tráfico ferroviario tiene dos grandes objetivos que son garantizar la seguridad y optimizar la explotación de la vía férrea. Para conseguir esto se emplean estos medios técnicos:

- Sistemas que garantizan la seguridad en el movimiento de los trenes (enclavamientos). Al tren que debe de ir de un punto de la vía férrea a otro, se le da paso si es posible y se impide que otros trenes invadan la ruta que se le ha asignado. El objetivo principal de estos sistemas es la seguridad.

- Sistemas de protección de los trenes (ATP, Automatic Train Protection), que indican al conductor del tren la velocidad máxima a la que debe de circular, y en caso de que la sobrepase, lo frenan. El conductor puede ser una persona o un sistema automático (ATO, Automatic Train Operation). Estos sistemas intervienen en la seguridad y en la explotación.

- Sistemas de gestión del tráfico para toda una línea (CTC o Control de Tráfico Centralizado), de acuerdo con los horarios de todos los trenes, su prioridad, y de la información del estado de la línea que proporcionan los sistemas de seguridad. Estos sistemas están orientados exclusivamente a mejorar la explotación.

La figura ofrece un esquema de estos diferentes tipos de sistemas y de las interfaces que existen entre ellos, representados por flechas.

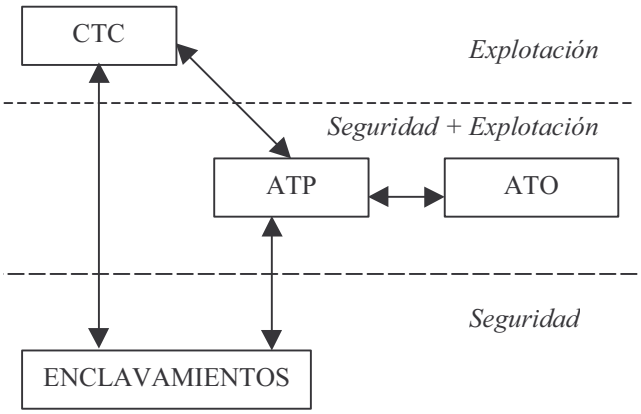


Figura 4.3 Sistemas de control ferroviario

4.1.1.1 Señales

Son dispositivos que están situados a lo largo de la vía férrea y que sirven para transmitir al maquinista avisos y órdenes. Existen señales ópticas y señales luminosas. Las señales ópticas son las más antiguas y están compuestas por uno o más brazos mecánicos que adoptan diferentes posiciones según lo que se quiera indicar. Las señales luminosas pueden tener una o varias lámparas de diferentes colores o del mismo color, que pueden estar encendidas o apagadas en función del mensaje que se quiera dar al tren. Se entiende por aspecto de una señal cada una de las diferentes combinaciones de lámparas encendidas que son posibles.

En el mundo existe una gran variedad de sistemas de señalización ferroviaria [Pa02][Pac02]. Se pueden clasificar en los siguientes tipos:

- Señales de velocidad: se le indica al tren una velocidad que no debe sobrepasar.
- Señales de ruta: las señales están situadas junto a las agujas e indican al tren el camino por donde se le va a llevar y es el maquinista quien tiene que saber cual es la velocidad límite de cada posible trayecto.

La mayoría de los ferrocarriles modernos usan las señales de velocidad. Las señales de ruta se usan en los ferrocarriles británicos y norteamericanos.

En algunos sistemas la información de ruta o de velocidad está directamente integrada en el aspecto de la señal, por ejemplo combinando lámparas de diferentes colores. Otros sistemas europeos más modernos emplean señales auxiliares como indicadores de ruta y de velocidad. En este tipo de sistemas, la señal denominada principal o de bloqueo sólo da información acerca de la ocupación de las siguientes secciones de bloqueo, de forma que los aspectos de la señal de bloqueo son relativamente simples. La mayoría de los ferrocarriles usan sólo tres aspectos

- Rojo – Parada.
- Amarillo – Proximidad, prepararse para parada en la próxima señal.
- Verde – Marcha.

RENFE emplea además entre otros aspectos de señal, el rojo blanco intermitente para itinerarios de baja velocidad, el rojo blanco para maniobras, el verde intermitente en vías de velocidad alta para avisar del verde y la combinación verde amarillo para indicar la proximidad de un desvío.

Existen diferentes tipos de señalización en cuanto a la noción de proximidad

- Señalización de bloqueo único.
- Señalización de bloqueo múltiple.

En la señalización de bloqueo único, la señal principal sólo da información de la sección de vía que hay a continuación, pero no da información sobre el aspecto de la siguiente señal. Por este motivo, cada señal principal está precedida por una señal de avanzada que avisa al maquinista del aspecto de la señal principal y que está situada a una distancia que permite frenar al tren antes de la señal principal si es necesario. Si las secciones de bloqueo son cortas, se monta la señal de avanzada en el mismo mástil que la señal de bloqueo.

En la señalización de bloqueo múltiple una señal de bloqueo da información sobre dos o más secciones de bloqueo que van a continuación. Las señales de avanzada no suelen ser necesarias en este tipo de sistema, pues se usa el aspecto de la señal de bloqueo para indicar el estado de la siguiente sección de vía.

Un ejemplo de este sistema es la señalización británica, que usa cuatro aspectos básicos

- Rojo – parada.
- Amarillo – proximidad.
- Doble amarillo – aviso de proximidad en la siguiente señal.
- Verde – Marcha.

Otro ejemplo de este tipo de sistemas es el bloqueo múltiple banalizado (con vía doble, pudiendo circular en los dos sentidos por ambas vías) que emplea RENFE en el corredor del Mediterráneo, donde la señal con aspecto verde va precedida por otra en verde intermitente.

Los sistemas de señalización de bloqueo múltiple permiten incrementar la velocidad de circulación, pero son poco efectivos si las secciones de bloqueo son mucho más largas que la distancia de frenado, pues entonces la información de proximidad se da demasiado pronto. Hay sistemas de señalización que pueden emplearse en modo de señalización de bloqueo único o en señalización de bloqueo múltiple: el sistema Ks alemán, el NORAC norteamericano que se usa en la parte este del país. Un sistema de señalización de bloqueo múltiple es el OSZD desarrollado por los ferrocarriles de la antigua Unión Soviética.

[Bai95] ofrece una descripción de los sistemas de señalización vigentes hoy en Europa. En cada país europeo se usan uno o varios sistemas de señalización diferentes y la única característica común a todos es que las señales en rojo indican parada.

Cuando las secciones de vía entre señales de bloqueo son más cortas que la distancia máxima de frenado, existen los dos siguientes tipos de solución:

- Colocar dos señales de proximidad por cada señal de bloqueo, de forma que los trenes, de cada dos secciones de vía, sólo pueden parar en una. Es una solución simple, pero que exige que las secciones de bloqueo tengan todas la misma longitud y que limita el número de circulaciones.

- Señalización de velocidad progresiva. El tren se va frenando paulatinamente según va pasando por las dos señales de proximidad. Es el método más flexible.

4.1.1.2 Bloqueos

Se denominan bloqueos o sistemas de bloqueo aquellos que sirven para garantizar la separación de los trenes en una determinada línea férrea que une dos estaciones. Los bloqueos se dividen según la responsabilidad que tiene el operador en manuales (todo lo hace un operador) y automáticos, (una máquina ayuda al operador). Ambos tipos deben detectar la presencia de otros trenes en la vía y activar las señales con el aspecto que corresponda en cada caso para dar paso al tren. La mayoría de los sistemas de bloqueo contemplan la posibilidad de que un tren pueda rebasar una señal a velocidad muy reducida sin que el operador deba de dar permiso expreso al maquinista para ello.

Bloqueo manual

Los operadores comprueban que el trayecto entre dos estaciones está libre anotando el número de cada tren que pasa por la estación y anunciando la llegada de dicho por teléfono a la estación adyacente, también llamada colateral. Cuando el tren rebasa una sección de bloqueo completa y esta protegido por una señal en rojo frente a una colisión por detrás, se notifica por teléfono a la estación de origen que esa sección de bloqueo se ha liberado. Todos los movimientos de trenes quedan reflejados en un libro de registro. En estaciones comunicadas por vía doble, este procedimiento se aplica sólo para movimientos de trenes en sentido contrario al predeterminado.

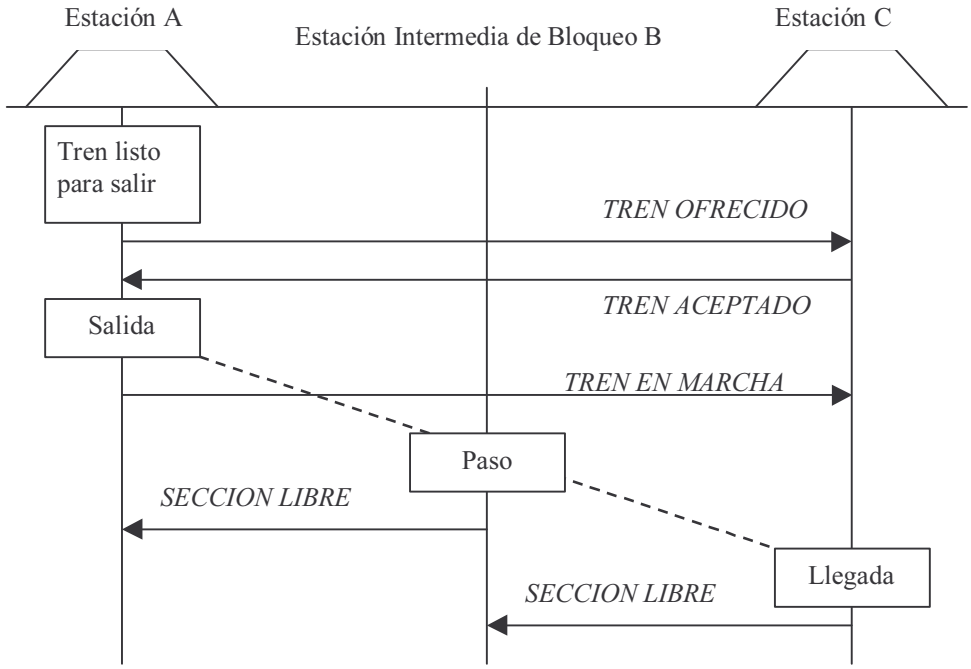


Figura 4.4 Bloqueo Manual

Este procedimiento se aplica en los ferrocarriles europeos tal cual se ha descrito previamente. En los ferrocarriles norteamericanos se da la variante consistente en que el control del bloqueo se realiza desde un puesto central, que va autorizando las distintas circulaciones de los trenes entre estaciones y recibiendo notificaciones de cada estación de cuando pasan los trenes por ellas. Esta variante optimiza mejor el tráfico y reduce el personal necesario, pero es viable sólo cuando la circulación de trenes es muy baja.

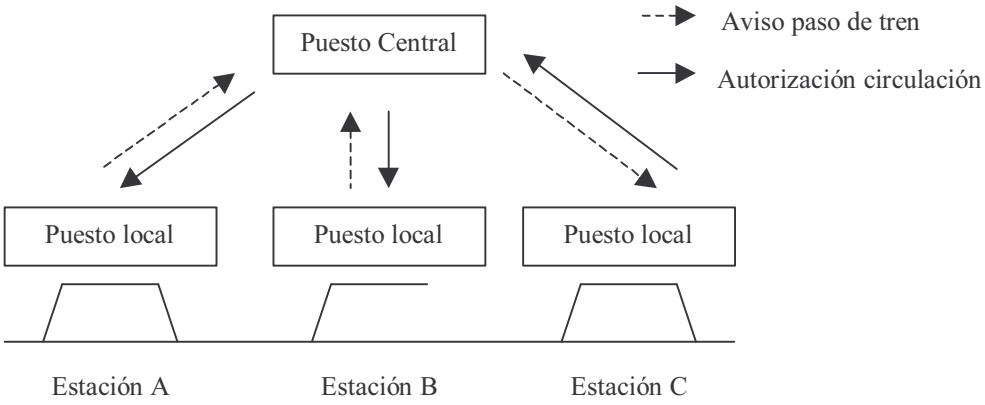


Figura 4.5 Control de Bloqueo por Operador Central

Para reducir el posible riesgo debido al error humano, hay sistemas de bloqueo manual que sustituyen la comunicación puramente telefónica por dispositivos electromecánicos. Los más sofisticados asocian la autorización de circulación con el aspecto de las señales entre las estaciones de tal forma que solo puede autorizar una circulación la estación colateral que no tiene las señales de salida en rojo.

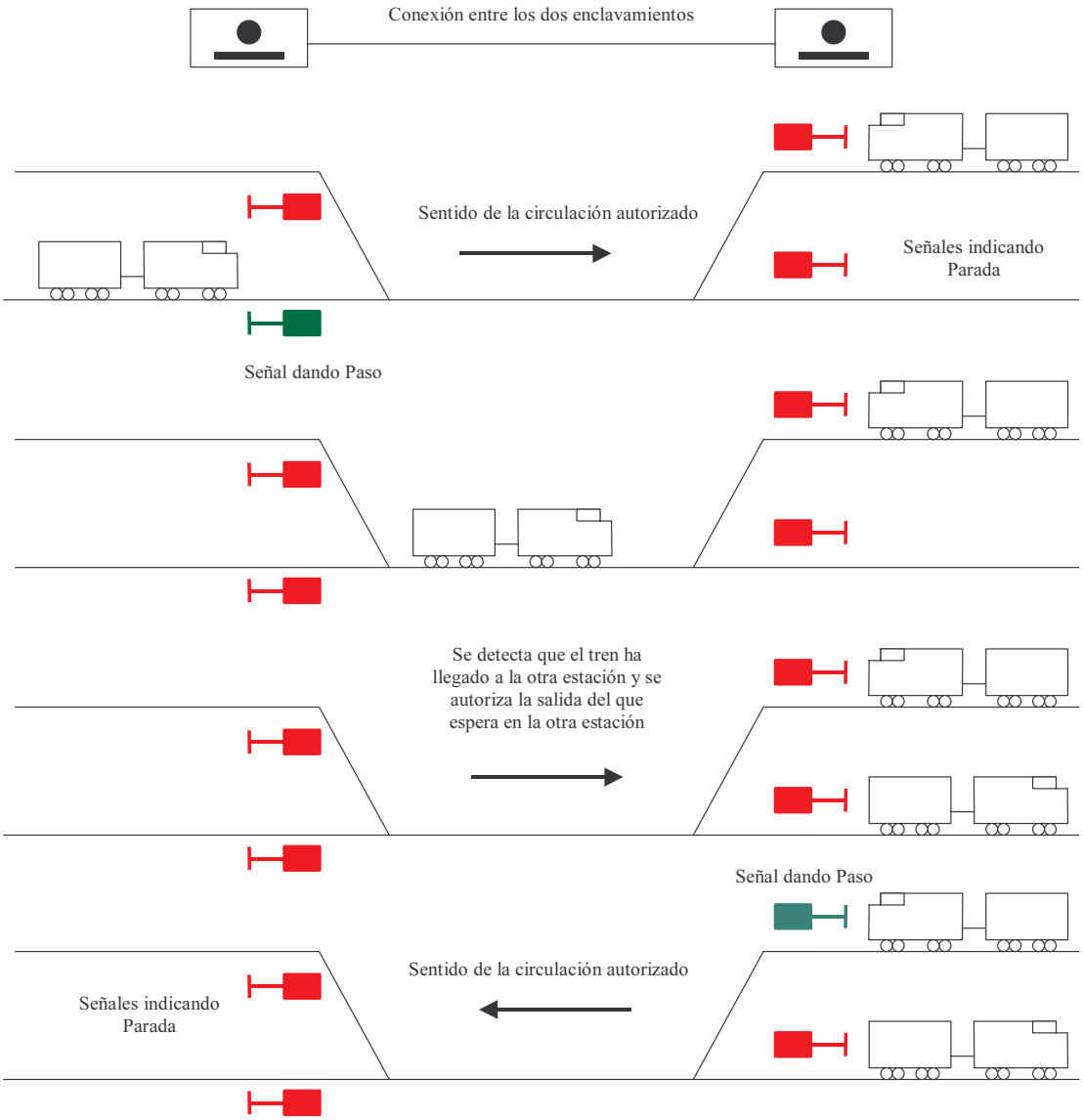


Figura 4.6 Funcionamiento simplificado de un bloqueo

Bloqueo automático

En este tipo de bloqueo, los operadores ya no están a cargo de comprobar visualmente si el trayecto está libre o no al autorizar una circulación de una estación a otra estación colateral, sino que dicha supervisión se realiza mediante un sistema de detección de ocupación de vía. El operador sólo se ocupa de autorizar el bloqueo (dar paso a los trenes) desde la estación de origen a la estación de destino y el sistema automático acepta la orden de autorización del operador comprobando un conjunto de condiciones que varían según el sistema y la estación, p.ej.:

- En la estación colateral no hay nada que impida autorizar el bloqueo,
- Las señales del trayecto de bloqueo se pueden poner en el aspecto requerido,
- Las secciones de vía correspondientes están libres, etc.

Para detectar si una sección de vía está ocupada por un tren, existen dos tipos de sistemas:

- Circuitos de vía: un circuito de vía es un circuito eléctrico construido con los raíles de una sección de vía, que coincide habitualmente con las secciones que marcan las señales. El circuito está alimentado por una fuente que puede ser de corriente continua o alterna y está cerrado por un equipo detector de paso de corriente, que suele ser un relé ferroviario de seguridad. Las secciones de vía están aisladas eléctricamente unas de otras. Cuando los ejes de un tren entran en la sección, interrumpen el paso de corriente en el equipo detector.

- Contadores de ejes: un contador de ejes es un sistema constituido por dos sensores de paso de un par de ruedas de un tren (un eje) situados a cada extremo de la sección de vía y un contador que está conectado a los sensores. El contador calcula la diferencia del número de ejes que han sido detectados por cada sensor, y si son diferentes, significa que hay un tren en la sección. Los sensores suelen estar duplicados para detectar el sentido de la marcha del tren y evitar que se produzcan errores en la cuenta si el tren se para y un eje queda justo a la altura del sensor.

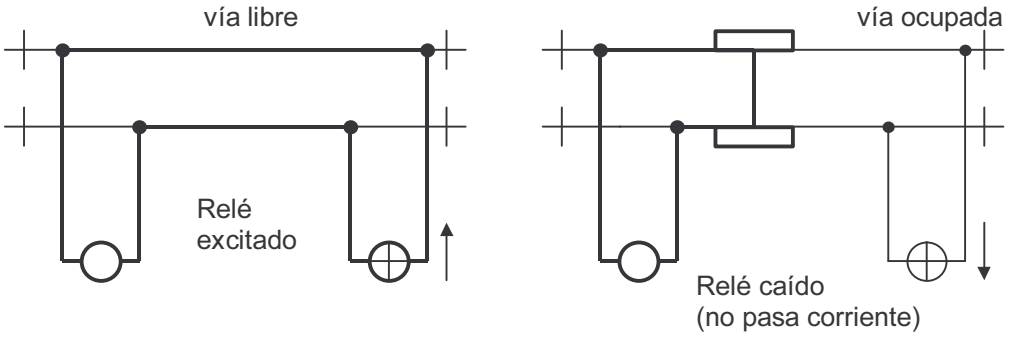


Figura 4.7 Esquema circuito de vía

4.1.1.3 Seguridad de las rutas ferroviarias: Enclavamientos

El enclavamiento es el sistema responsable de garantizar la seguridad en los movimientos de los trenes dentro de una estación mediante el mando y supervisión de las agujas, las señales y los demás elementos relevantes para la seguridad, como por ejemplo, los circuitos de vía, los pasos a nivel, etcétera. El enclavamiento está a cargo de un operador, que es quien manda la ruta por la que quiere llevar a cada tren. Si la ruta puede ser establecida sin peligro, el enclavamiento lleva a cabo el mando del operador moviendo las agujas a la posición necesaria y dando paso al tren mediante los aspectos de las señales.

El enclavamiento es un sistema local a la estación y tiene por tanto unas fronteras. Puede tener comunicación con otros enclavamientos colaterales, directamente o a través de un sistema de bloqueo.

La seguridad en una ruta ferroviaria (itinerario) se da si se cumplen las siguientes condiciones:

- Las agujas están en la posición correcta y enclavadas, es decir, no se pueden mandar mover a la otra posición.
- Los itinerarios incompatibles están bloqueados. Se define como incompatibles a los itinerarios que si están activados a la vez provocan una situación peligrosa, como por ejemplo, choque frontal, trasero o lateral de dos trenes.
- El trayecto está libre.

Esto está garantizado mediante los siguientes principios:

- Dependencia de agujas y señales.
- Bloqueo de itinerario
- Incompatibilidad de itinerarios.
- Protección de flanco.
- Detección de vía ocupada.

Estos principios se aplican de forma menos restrictiva para las maniobras, ya que son una situación en la que los trenes se mueven muy despacio bajo la responsabilidad del jefe de circulación y del maquinista. Algunos ferrocarriles utilizan unas rutas específicas de maniobra con sus aspectos de señal propios (itinerarios de maniobra). Puede haber señales que sean exclusivamente para maniobras.

Un itinerario comienza siempre en una señal gobernada por el enclavamiento. El final del itinerario puede ser otra señal del enclavamiento, o en caso de no existir esta señal de final (p.ej. si el itinerario es de salida, es decir termina en la frontera del enclavamiento) la sección de vía que hay tras la última aguja. En los ferrocarriles norteamericanos, la disposición de las señales en las fronteras del enclavamiento difiere de la de los ferrocarriles europeos.

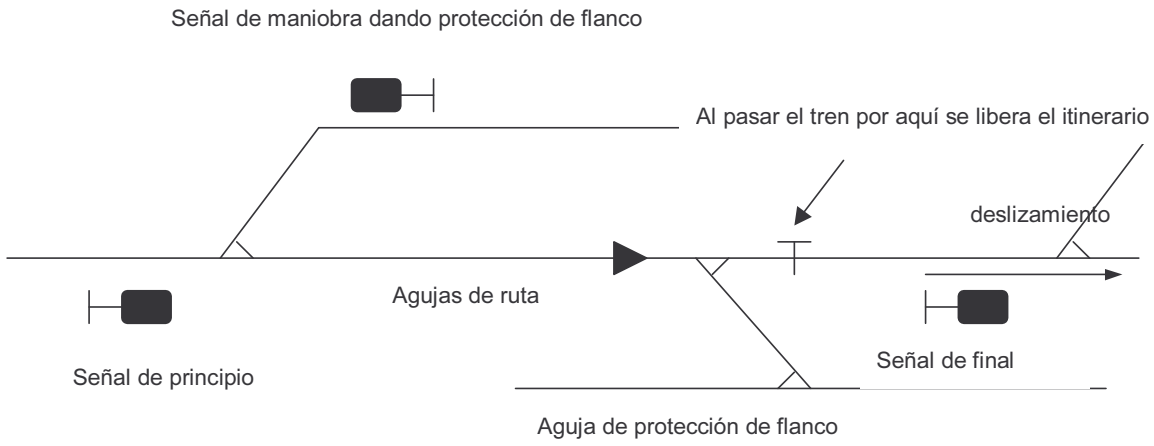


Figura 4.8 Elementos de un itinerario

A continuación se explican con más detalle los principios de seguridad de los itinerarios que se han enumerado antes.

Dependencia de agujas y señales

Consiste en lo siguiente:

- Una aguja no se puede mover mientras las señales estén dando paso.
- Una señal no puede abrirse si la aguja no está en la posición requerida y comprobando. Por seguridad una señal siempre se puede cerrar (poner en rojo). La comprobación de una aguja significa que la aguja está en una de las dos posiciones que permiten que el tren pase por ella sin descarrilar.

El bloqueo de itinerario

Se produce cuando el operador del enclavamiento manda establecer el itinerario y el enclavamiento acepta dicho mando, lo cual produce que el trayecto del itinerario y sus elementos de ruta (agujas, señales, etc.) queden reservados en exclusiva para dicho itinerario hasta que este se libere.

Para disolver (liberar) el itinerario se requiere que un tren circule el itinerario desde el principio hasta el final o que el operador mande disolver el itinerario. Si hay un tren dentro del trayecto, se necesita habitualmente un mando especial de emergencia para ello. Si el tren no ha entrado aun en el trayecto, disolver el itinerario se puede hacer directamente sin hacer el mando de emergencia.

Se denomina deslizamiento al espacio de seguridad que hay tras la señal de final de algunos itinerarios y que suele ir hasta la aguja inmediatamente posterior a la señal de final. Cuando el tren ha circulado por completo el itinerario, se espera un tiempo hasta que el deslizamiento se libera también.

Incompatibilidad de itinerarios

Consiste en que el enclavamiento rechaza que se establezcan simultáneamente dos o más itinerarios que entran en conflicto, con el fin de prevenir choques frontales, traseros, o laterales de los trenes. Los itinerarios incompatibles se cruzan o se solapan.

Protección de flanco

Consiste en impedir que entren en una determinada vía trenes que circulan por una vía adyacente. Esto puede llevarse a cabo o bien por reglas de explotación, que está sujeto al fallo humano o mediante las agujas, y las señales que las protegen. También se puede usar para protección de flanco una variante de la aguja que es el calce. El calce es una sección de vía móvil con dos posiciones, levantado (no permite el paso del tren por encima de él) y abatido. Se colocan habitualmente fuera de las vías principales de la estación.

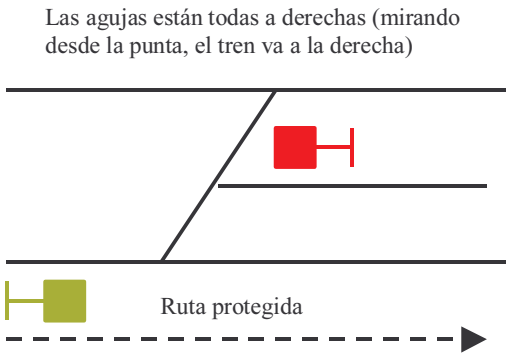


Figura 4.9 Protección de flanco

Detección de vía ocupada

Los enclavamientos actuales están dotados de sistemas de detección de vía ocupada, ya sea con circuitos de vía o con contadores de ejes. A la hora definir las secciones de vía hay que tener en cuenta que no se puede ni hacerlas muy grandes (se admiten pocos trenes circulando simultáneamente) ni muy pequeñas (secciones no protegidas por señales).

Variedades de enclavamientos

Existen diferentes variedades de enclavamientos según la tecnología con la que están construidos. Los enclavamientos mecánicos, que son los más antiguos (siglo XIX), funcionan basándose en palancas y poleas con las que se mueven las agujas y se abren y cierran las señales de tipo mecánico (no luminosas). El nombre de enclavamiento viene precisamente de estos sistemas, pues para formar un itinerario, hay que poner las palancas asociadas a sus elementos en la posición requerida y luego bloquearlas (enclavarlas) mecánicamente con una barra específica del itinerario. A los enclavamientos mecánicos les sucedieron en parte los enclavamientos hidráulicos, en los que se sustituyen los cables mecánicos por agua a presión para poder manejar agujas y señales a mayor distancia del puesto de control.

Los enclavamientos eléctricos (1900) usan la misma filosofía de interbloqueo mecánico que los enclavamientos mecánicos, pero accionando las agujas y las señales eléctricamente en lugar de usar palancas. Posteriormente (en torno a 1920) se inventaron los enclavamientos de relés, que usan relés electromagnéticos para realizar la lógica de enclavamiento y accionar las señales luminosas y agujas conectadas a motores. Cada itinerario es un circuito eléctrico que para cerrarse necesita tener todos los interruptores asociados a cada elemento de ruta dando paso.

La aparición de la electrónica y el software ha propiciado que los sistemas de relés se hayan reemplazado parcial o totalmente por ordenadores especializados, que son los llamados enclavamientos electrónicos (1980). El ritmo de renovación de los enclavamientos que hay en servicio es muy variable según el país y la instalación en particular y es posible hoy en día encontrar enclavamientos en servicio que emplean tecnologías antiguas.

Otra clasificación de enclavamientos se hace en función de cómo está diseñada internamente la lógica del sistema. Se tiene así los enclavamientos basados en tablas o tabulares y los enclavamientos geográficos.

Los enclavamientos tabulares basan su lógica de interbloqueo de agujas y señales en una tabla que contiene las condiciones de interbloqueo para todos los itinerarios. Dentro de los enclavamientos tabulares, puede haber enclavamientos de lógica en cascada (Norteamérica y Gran Bretaña) y enclavamientos definidos basándose en incompatibilidades (Europa continental). Los enclavamientos de lógica en cascada definen para cada itinerario una secuencia fija de estados (posiciones de las agujas y aspectos de señal) en los elementos del itinerario hasta establecerlo (abrir la señal de entrada). Los enclavamientos definidos basándose en incompatibilidades se definen con una tabla como la que hay a continuación.

ITINERARIOS	ITINERARIOS COMPATIBLES			Posición agujas asociadas	
	A	B	C	Aguja 1	Aguja 2
A	-	SI	SI	+	+
B	SI	-	NO	+	-
C	SI	NO	-	-	-

Los enclavamientos geográficos implementan una lógica basada en la situación de los elementos del itinerario. Una vez que el operador dice el principio y el final del itinerario, el sistema parte de la señal de principio de itinerario y va comprobando que el estado es correcto en las agujas y señales intermedias hasta llegar a la señal de final, momento en el cual, la señal de principio se abre.

4.1.1.4 Supervisión de la velocidad de los trenes: Protección Automática de Tren (ATP)

Los sistemas ATP transmiten información sobre las autorizaciones de circulación (hasta donde) y los límites de velocidad desde la vía al tren y pueden frenar la marcha del tren si por cualquier razón este sobrepasa los límites que le son establecidos. Donde existen señales a lo largo de la vía, el ATP recibe

información del estado de las señales y supervisa que los trenes no rebasen las señales si están en rojo. En trenes que disponen de señalización en la cabina del maquinista, la información relevante para la marcha del tren es transmitida por el sistema ATP.

Hay dos tipos de sistemas ATP, según el modo de transmisión utilizado entre la vía y el tren, ATP intermitente y ATP continuo.

ATP Intermitente

En estos sistemas los datos se transmiten de la vía al tren sólo en determinados puntos. Las formas de transmisión pueden ser las siguientes, citadas de mayor a menor antigüedad:

- Aparatos mecánicos.
- Contactos eléctricos.
- Inducción electromagnética.
- Transpondedores con transmisión digital.

Un ATP puede tener tres funciones principales:

- Sistema de aviso automático.
- Supervisión de curva de frenado.
- Parada de tren.

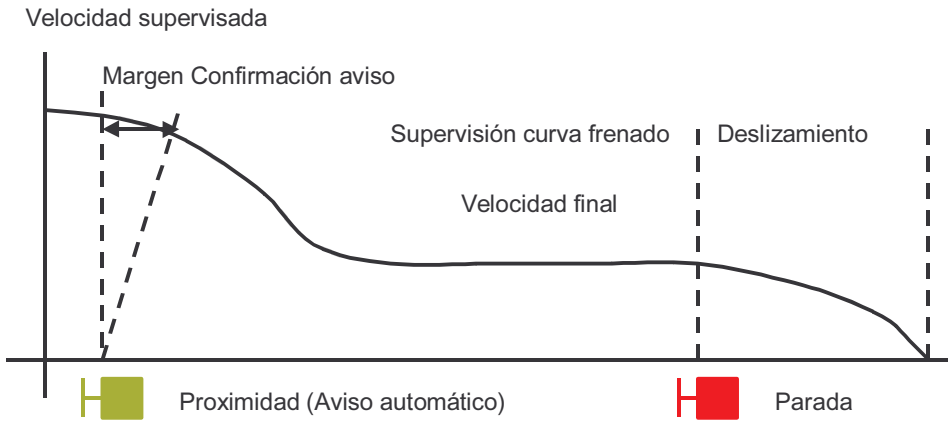


Figura 4.10 Funcionamiento de un ATP intermitente

Al rebasar la señal de proximidad (la que hay antes de la que esta en rojo) el sistema de aviso automático advierte al tren de que debe de comenzar a frenar. Es usual que se necesite un acuse de recibo por parte del maquinista, que en caso de no recibirse, para automáticamente el tren. Al comenzar la frenada, el ATP controla en todo momento que se están respetando los límites de velocidad de la curva de frenado. La curva de frenado suele terminar marcando una velocidad muy baja que permita parar el tren de forma segura si es necesario y puede, según el tipo de ATP, ser fija o variable, dependiendo de los siguientes factores:

- Datos relativos al tren.
- Aspecto de la señal de proximidad.
- Perfil de inclinación de la vía, etc.

Un ejemplo de sistema ATP intermitente es el estándar ETCS (European Train Control System) de Nivel 1, que esta siendo impulsado dentro de la Unión Europea para facilitar el tránsito ferroviario entre los diferentes países. En este sistema, la transmisión de las informaciones hacia el tren se realiza mediante balizas. Hay dos tipos de balizas, fijas y conmutables. Las balizas fijas no tienen interfaces con otros equipos y transmiten siempre los mismos datos de posición al tren. Las balizas conmutables reciben información del enclavamiento, sobre los aspectos de las señales así como las rutas establecidas para que el ordenador de a bordo de la locomotora pueda calcular la curva de velocidad correspondiente. Adicionalmente permiten el establecimiento de restricciones temporales de velocidad.

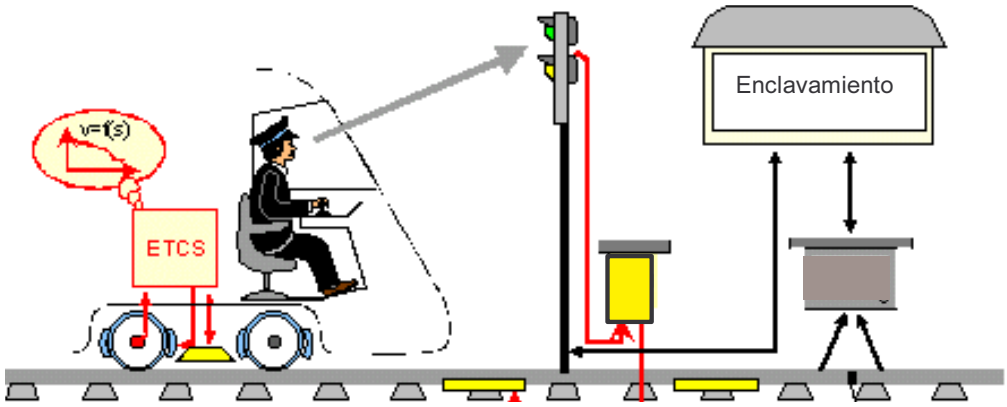


Figura 4.11 ETCS nivel 1

ATP Continuo

En este tipo de ATP, la velocidad y la posición del tren se supervisan constantemente. Se utilizan estas tres técnicas, citadas por orden de antigüedad:

- Circuitos de vía codificados.
- Transmisión de datos usando un cable tendido a lo largo de toda la vía como antena.
- Transmisión por radio combinada con balizas que el tren usa para conocer su posición.

La línea del AVE de Madrid a Sevilla usa el ATP continuo, denominado LZB en alemán (Linien Zug Beeinflussung) o CAT en español, (Conducción Asistida de Trenes) que presenta al maquinista, de forma continua, toda la información necesaria para la conducción.

Los sistemas de señalización ferroviaria con las señales al borde de la vía, no pueden emplearse en los trenes de alta velocidad, ya que a partir de velocidades superiores a 200 Km/h, la percepción y capacidad de reacción del maquinista no garantizan la seguridad. Este sistema tiene una antena de captación y emisión debajo de cada locomotora, la cual se comunica con un puesto de mando centralizado a través de una pareja de cables radiantes de comunicaciones situados entre los carriles. El enlace ascendente desde los cables radiantes hacia los trenes, se realiza a una frecuencia diferente que el enlace descendente entre el tren y la vía. A la cabina de conducción llegan múltiples informaciones, siendo las fundamentales la Velocidad Meta y la Distancia Meta, reportadas a la cabina del tren con una antelación de 12 Km. El equipo LZB soporta dos sistemas de conducción a voluntad del maquinista: el manual y el automático. Con el sistema automático, existen a su vez dos maneras de realizar la conducción: la primera con velocidad de crucero prefijada por el maquinista y la segunda manera en modo piloto automático a cargo del LZB y el maquinista se limita a la vigilancia de incidencias.

Otro ATP continuo es el sistema estándar ETCS (European Train Control System) nivel 2. Para conocer su posición los trenes utilizan la información que reciben de las balizas colocadas a lo largo

de la vía. Existe un puesto de mando central, el RBC (Radio Block Center) que es quien determina la Velocidad Meta y la Distancia Meta y las transmite al tren correspondiente a través del sistema de radiotelefonía GSM-R.

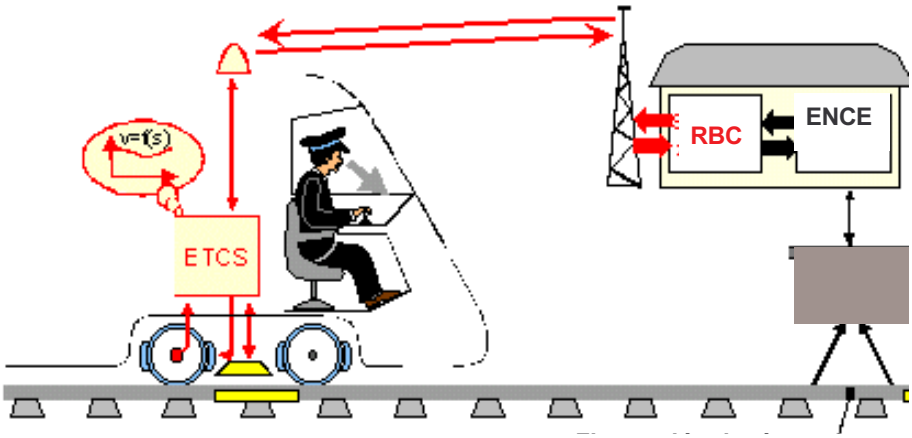


Figura 4.12 ETCS Nivel 2

4.1.1.5 Gestión del tráfico ferroviario

Control local

- Líneas férreas sin señales

En este tipo de líneas, la gestión del tráfico está en manos de un gestor central que por teléfono, va dictando instrucciones a los maquinistas. Este tipo de gestión es el habitual en Norteamérica. En lugar del teléfono, también puede usarse la radio como medio de comunicación con los trenes. Estos sistemas ponen en el gestor toda la responsabilidad y no son adecuados para líneas con mucho tráfico.

- Líneas férreas con señales

En esta configuración, las señales y las agujas están gobernadas por enclavamientos, y cada enclavamiento está manejado por un operador. Los operadores se comunican por teléfono entre sí y con un gestor central, que registra por escrito el paso de los trenes por cada estación. En Norteamérica este registro tiene forma de tabla, mientras que en el resto de los ferrocarriles se usa un formato gráfico. El gestor puede por medio de este registro detectar posibles problemas y tomar decisiones sobre el tráfico, como por ejemplo, cambiar la secuencia de paso de los trenes.

Este gestor central tiene diferente nivel de responsabilidad según los países. En Norteamérica, es quien tiene la máxima responsabilidad, pues es el que da las autorizaciones de circulación a los trenes. En los ferrocarriles europeos esta responsabilidad está a cargo de los operadores locales, aunque también varía dentro de los diferentes países. Por ejemplo en Alemania, hay puestos locales que a su vez mandan sobre otros puestos locales, mientras que en Inglaterra, todos los puestos locales tienen igual responsabilidad.

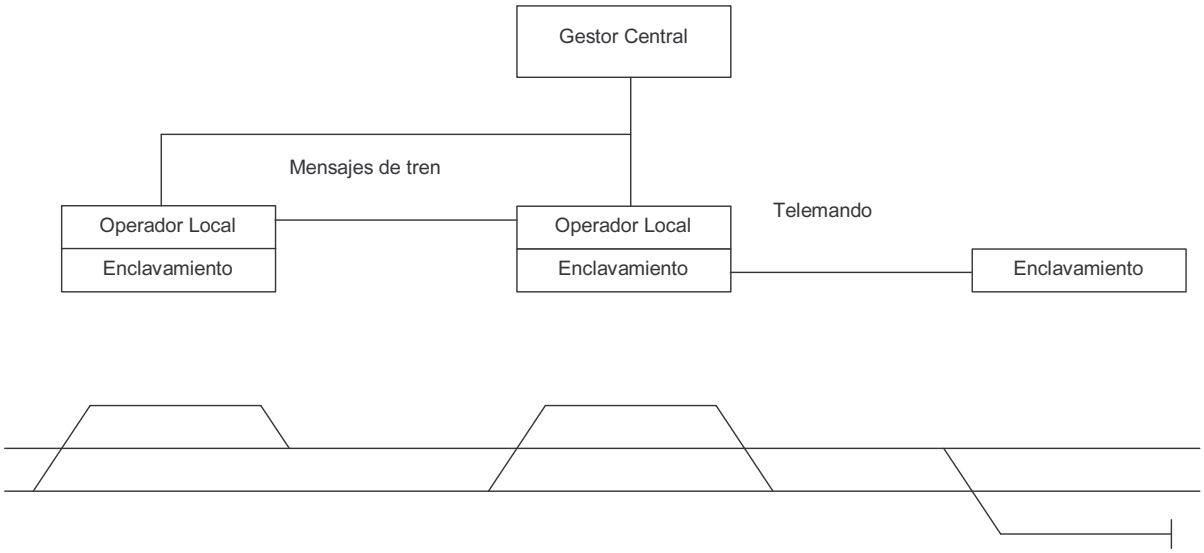


Figura 4.13 Control de tráfico en una línea con operadores locales y operador central

Control de tráfico Centralizado (CTC)

El CTC permite controlar las agujas y señales de un conjunto de enclavamientos situados a lo largo de una línea ferroviaria. Estos enclavamientos no tienen operador local salvo en casos de emergencia. Este sistema se usa habitualmente en países con líneas férreas largas que atraviesan zonas con poca densidad de población, por ejemplo, Norteamérica, Rusia, y también España. O también puede usarse en líneas de mucho tráfico para descargar a los operadores locales de trabajo.

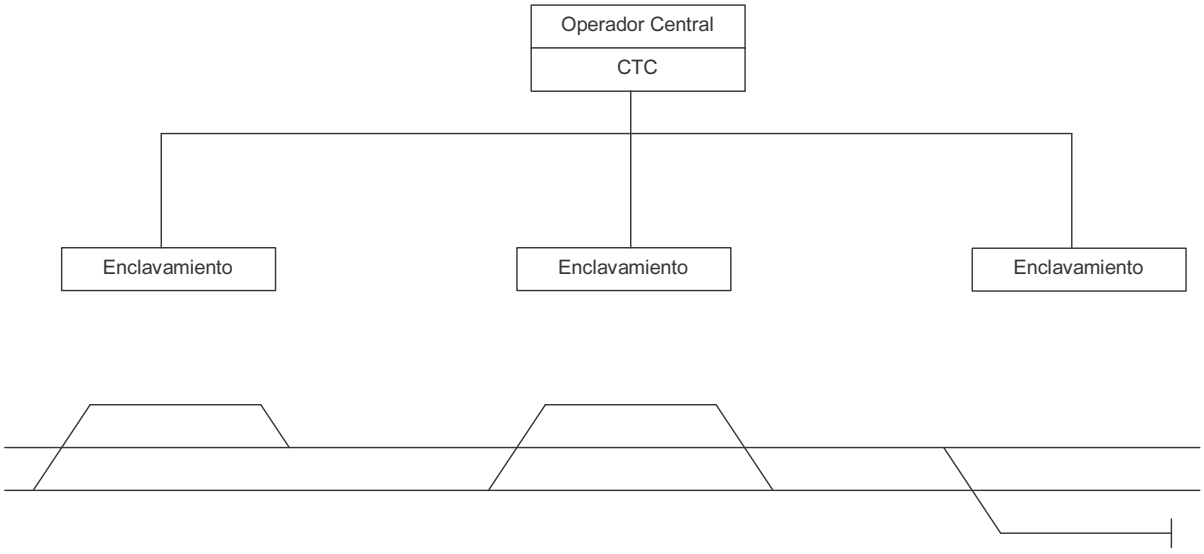


Figura 4.14 Control de tráfico centralizado (CTC)

El CTC conoce todos los trenes que circulan por la línea en cada momento y puede dar prioridad a unos trenes frente a otros. En este tipo de sistemas existen ejemplos de aplicación de técnicas de ingeniería software avanzadas como por ejemplo, las redes neuronales o la inteligencia artificial [Fri92][Ru98].

Sistemas Auxiliares

- Descriptores de trenes

Estos sistemas muestran todos los trenes que circulan por la línea y son considerados "no seguros", es decir que pueden fallar sin obligar a la parada de los trenes. Pueden obtener la información sobre la posición de los trenes de las siguientes formas:

  - Información de las secciones de vía ocupadas de los enclavamientos.
  - Identificadores de los trenes leídos por equipos situados en la vía.
  - Detección de posición realizada por el propio tren por medio de GPS o leyendo balizas indicadoras de posición situadas en la vía.

- Formación Automática de Itinerarios (FAI)

Estos sistemas solicitan al enclavamiento que establezca una ruta cuando un tren se aproxima, con el fin de evitar que el tren tenga que esperar a que el jefe de circulación mande establecer la ruta manualmente. Un sistema FAI tiene las siguientes funciones:

a) Selección automática de una ruta por proximidad del tren a una aguja o señal determinadas. Esta selección puede realizarse según una secuencia predeterminada común a todos los trenes, o identificando a cada tren y accediendo a una base de datos que contiene la ruta para cada tren.

b) Mando de la ruta en el momento adecuado: ni demasiado pronto, para no bloquear innecesariamente parte de la estación en prejuicio de otros trenes, ni demasiado tarde, para no obligar al tren para el que se solicita la ruta a frenar.

c) Resolución de conflictos entre rutas incompatibles. Esta tarea puede realizarse usando o bien una base de datos con los horarios de todos los trenes, o una base de datos con los destinos de todos los trenes. En el primer caso, los trenes que van con retraso respecto de su horario tiene prioridad. En el segundo caso, antes de mandar al enclavamiento establecer una ruta, el sistema FAI comprueba si esa ruta es incompatible con las rutas que va a solicitar para los otros trenes, con el fin de evitar posibles colapsos en la circulación.

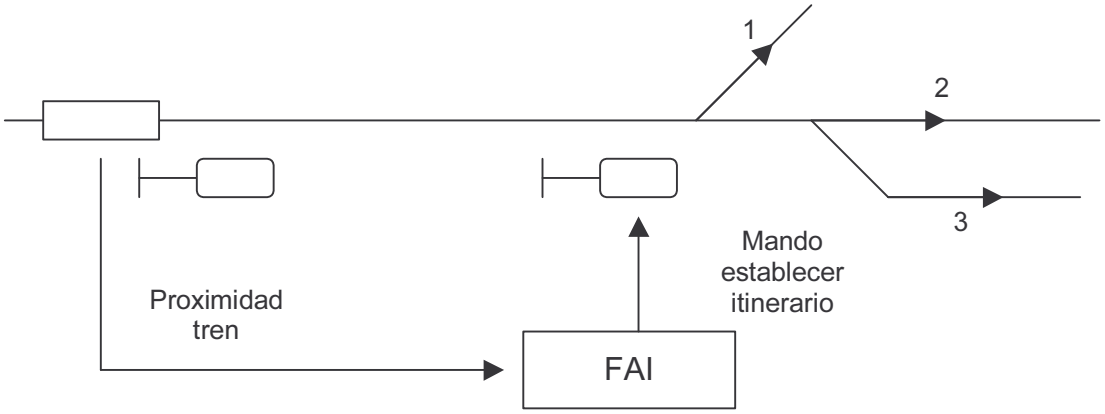


Figura 4.15 Formación Automática de itinerarios (FAI)

4.1.1.6 Iniciativas de estandarización

Todo lo dicho hasta ahora sobre los sistemas de control ferroviario va encaminado a dar una idea de su complejidad y de las variedades que presentan. Para abordar esta variedad, las Líneas de Producto Software son una estrategia válida, pero también son importantes todos los esfuerzos encaminados a lograr una cierta homogeneidad. Esto en Europa es un problema especialmente grave por la cantidad de diferencias que hay entre países, lo cual encarece y dificulta el tráfico ferroviario internacional. Ejemplos de estándares ferroviarios europeos son los siguientes:

- Las normas CENELEC [CEN97], que permiten obtener certificaciones de sistemas ferroviarios válidas a escala europea, lo cual es interesante para las compañías multinacionales del sector. Aún se encuentran en fase de maduración, pues las diferencias en el ferrocarril son demasiado significativas en Europa todavía. Por ello las certificaciones nacionales tienen más importancia actualmente. Un ejemplo de aplicación de estas normas, concretamente la 50128 al proceso de desarrollo software está explicado en [Bre98]. En la figura se ve el conjunto de las distintas partes de la norma CENELEC y sus ámbitos respectivos.

- El estándar ERTMS/ETCS [UN00] para sistemas interoperables ATP de tipo continuo (Nivel 2) y ATP discontinuo (Nivel 1). Este estándar de señalización es el preferido para la instalación de AVE de Madrid a Barcelona. En España hay una vía piloto de ERTMS en Albacete.

- El estándar EUROINTERLOCKING [Koe01] pretende normalizar los requisitos funcionales y las interfaces de los enclavamientos, y está todavía en su fase inicial.

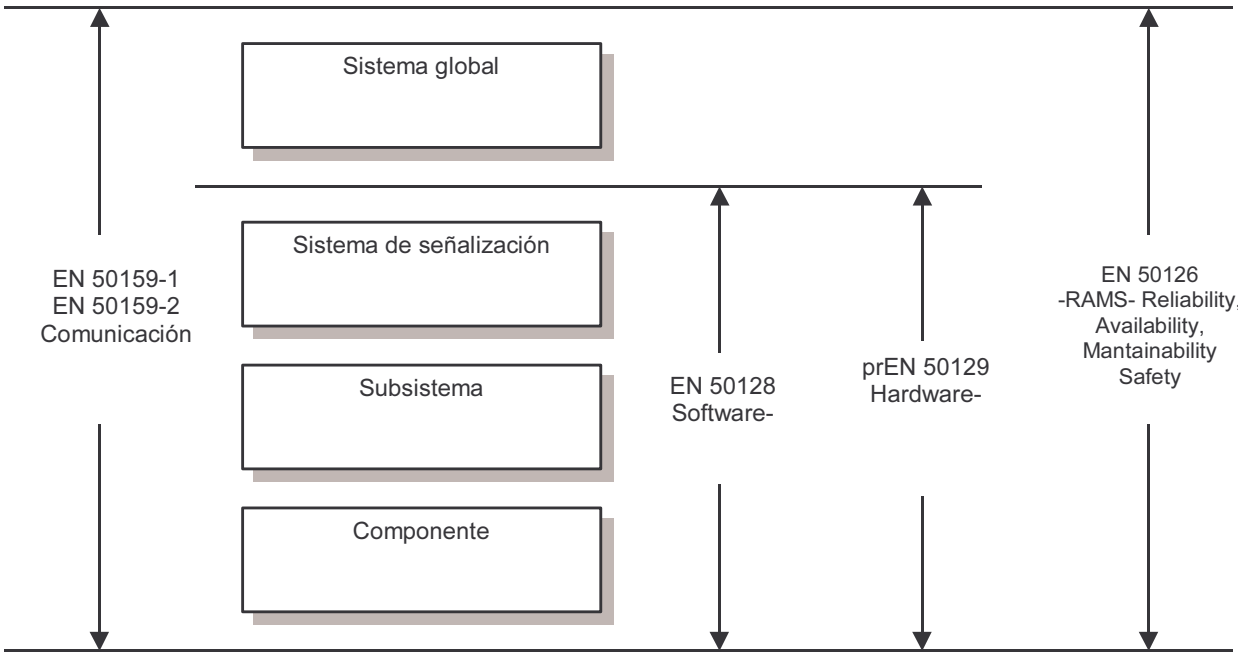


Figura 4.16 Estándares CENELEC

4.1.2 La validación de sistemas de control del tráfico ferroviario

Un fallo de software en un sistema de control ferroviario puede acarrear graves riesgos tanto para los trenes como para los pasajeros. Por eso las pruebas de validación de estos sistemas necesitan ser especialmente metódicas. La norma CENELEC [CEN97], reconocida en la Unión Europea como estándar de calidad para sistemas ferroviarios, define la validación como la constatación de que un producto satisface sus requisitos. Las pruebas de validación se hacen antes de entregar el software al cliente para probar en la instalación real. Un error no detectado en la validación, aparecerá, en el mejor de los casos durante las pruebas de aceptación con el cliente, o peor, durante el funcionamiento normal del sistema.

La validación de software en el dominio del control de tráfico ferroviario es cada vez más compleja. Los sistemas siguen teniendo los mismos requisitos de seguridad, pero la tecnología actual ofrece herramientas más poderosas y los plazos de puesta en el mercado se han acortado. Por otro lado, las normas de calidad (ISO, CENELEC) han cambiado también la forma de desarrollar software, exigiendo un mayor esfuerzo en la documentación y verificación del proceso de desarrollo.

La falta de tiempo a causa de los plazos es habitualmente un problema grave para la validación de los sistemas software de control ferroviario. El gran esfuerzo que supone la validación debe repetirse para las distintas versiones de un mismo sistema de control ferroviario. Si se tiene en cuenta que los diferentes requisitos de los distintos clientes producen diferentes versiones del mismo sistema software, aun teniendo las diferentes variaciones una base común, el número de validaciones puede llegar a ser muy alto.

Es habitual que el cliente fije un conjunto obligatorio de pruebas para realizar sobre el sistema, pero en último término el responsable de que ese sistema funcione correctamente es la compañía que lo suministra. Aunque sea una nueva versión de software con un cambio muy pequeño, hay que hacer unas pruebas de regresión que permitan estar seguros de que el nuevo software es correcto.

Las estrategias que permitan optimizar este esfuerzo son por ello muy importantes. Si se realiza un esfuerzo para gestionar las diferentes variaciones en un sistema software constituyendo una línea de productos, un esfuerzo análogo en la validación merece también la pena. Este esfuerzo tiene aún más sentido si la automatización de las pruebas es posible. Las técnicas de automatización de pruebas reducen el esfuerzo en las pruebas de validación. La cuestión es tener unas pruebas automatizadas que a la vez sean fiables, es decir, que si hay un fallo este se detecte.

Los sistemas ferroviarios se han probado hasta la fecha de forma funcional. La incorporación de software ha hecho que los sistemas sean más flexibles y potentes, pero también más complejos y sujetos a fallos. Las normas de calidad han incidido en este problema, reglamentando el ciclo de vida del software. Para la validación, esto significa que hay que documentar qué pruebas se han realizado y qué requisitos se han validado mediante esas pruebas. Este es el primer paso para poder decir qué porcentaje del sistema se ha probado.

Tener un proceso de pruebas definido y documentado es la condición previa para que algunas de las pruebas puedan realizarse por personas menos expertas o de forma automática. No es la panacea definitiva, pues el conocimiento de una organización siempre está en las personas.

El estándar CENELEC exige como requisitos obligatorios las pruebas funcionales (de caja negra) y de rendimiento. La norma deja abierta la posibilidad de realizar otros tipos de prueba. La estrategia descrita en estas páginas va en esta dirección, complementando las pruebas de caja negra hechas de forma manual y las pruebas de rendimiento, incrementando así la calidad del proceso de prueba.

4.2 El caso de estudio: pruebas de validación de enclavamientos electrónicos

Este apartado presenta la estrategia de pruebas automatizadas utilizada por Alcatel en la validación de una línea de productos de control del tráfico ferroviario, concretamente, de enclavamientos electrónicos. La estrategia de validación empleada está definida por los dos puntos clave siguientes: un método que define un proceso de trabajo desde los requisitos hasta la realización de las pruebas, y la reutilización de la estrategia en los diferentes miembros de la línea de productos.

Lo expuesto en esta parte de la Tesis Doctoral refleja el trabajo del autor en la división de Automatización del Transporte de Alcatel como ingeniero de desarrollo de software y como ingeniero de pruebas de enclavamientos electrónicos para diferentes administraciones ferroviarias. El hecho de que se relate una aplicación práctica da un mayor interés a la Tesis Doctoral, pero hay que aclarar que cuanto se dice aquí en modo alguno representa una violación de aspectos confidenciales, puesto que lo que se cuenta ha sido ya publicado con el consentimiento de la compañía, ya sea por parte del autor de la Tesis Doctoral o por otros colegas [Lai00], [MeDu00], [MeDu01], [EMM02], [MET02], [Val00], [Boe00].

4.2.1 La línea de productos de sistemas de control del tráfico ferroviario validada

Antes de describir la estrategia de automatización de pruebas, es importante tener una idea global de la arquitectura del sistema en general, a partir de la que se derivan los diferentes productos y de su modelo de ciclo de vida software, siguiendo las ideas de [Cle01].

4.2.1.1 Aspectos comunes del sistema

La Figura 4.17 representa la arquitectura del enclavamiento, compuesta de los módulos OM, IM, DS y FEC. Las líneas entre los módulos indican los canales de comunicación. El flujo de información en un canal es bidireccional.

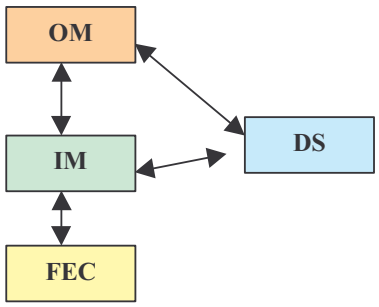


Figura 4.17 Estructura del enclavamiento completo

- OM (Operating Module, Módulo de Operación). Este módulo actúa como interfaz hombre máquina. Muestra el estado de la estación en un monitor videográfico que le notifica el IM y valida las órdenes que los operadores introducen en el sistema por medio de diferentes periféricos (teclado, ratón, o tablero gráfico) antes de mandarlas al IM. El operador puede ser una persona o un telemando (CTC) al que esté conectado el OM. La representación gráfica de la estación se rige

por un estándar definido por el cliente correspondiente, que en el caso de RENFE se denomina "Norma Videográfica". Este módulo no es crítico para la seguridad y por eso no tiene redundancia de hardware, pues si falla lo que sucede es que el operador no puede mandar itinerarios, pero el sistema sigue supervisando que los trenes se mueven de forma segura. Hay algunas administraciones ferroviarias, como por ejemplo la alemana, que requieren que la representación gráfica sea validada mediante un ordenador seguro (con hardware redundante), pero estos requisitos no existían a la hora de definir la estrategia de validación y por eso el OM no se ha contemplado en ella. El OM tiene una parte fija válida para todas las estaciones y otra parte configurable por medio de los datos de aplicación donde se definen los distintos elementos, los mandos posibles y la representación gráfica de la estación.

- IM (Interlocking Module, Módulo de lógica de enclavamiento). Este módulo es el "cerebro" del enclavamiento. Es el responsable de la seguridad de acuerdo con los principios expuestos en el apartado 4.1.1.3 titulado "Seguridad de las rutas ferroviarias: Enclavamientos" y está diseñado según el concepto de enclavamiento basado en tablas. El IM establece los itinerarios en sus diversas modalidades (por ejemplo en RENFE, itinerarios, maniobras y rebases autorizados), para lo cual recibe el mando del OM, y si el itinerario no es incompatible con los que tiene ya establecidos, manda al FEC mover las agujas de ruta y de protección de flanco que hagan falta, y posteriormente manda abrir al FEC la señal de principio de itinerario. También es responsable de la dependencia entre agujas y señales y del tratamiento de vías ocupadas (disolución de itinerario, cierre de señal, etc.). Implementa la lógica de bloqueo con los enclavamientos vecinos. Todo lo que afecta a la seguridad pasa por el control del IM: interfaz con el sistema ATP, Formación Automática de itinerarios, autorización del movimiento de agujas solicitado por medio de pulsador de maniobra local, control de pasos a nivel, etc. Por todo ello, el IM es uno de los módulos considerados dentro de la estrategia de validación. El IM tiene una parte fija válida para todas las estaciones y otra parte configurable por medio de los datos de aplicación donde se definen los distintos elementos de itinerario y su comportamiento, el número y tipo de las interfaces externas, y los itinerarios existentes de la estación.

- FEC (Field Element Controller, Módulo de Control de Elementos de Campo). Controla los elementos hardware externos, también llamados elementos de campo, que sirven para asegurar los movimientos de los trenes agujas, señales, etc. Un enclavamiento habitualmente tiene un único IM y varios FEC, que pueden estar separados físicamente. Recibe las órdenes que le llegan del IM para los elementos y notifica su estado al IM. El FEC gobierna los elementos de ruta por separado, es decir el IM manda al FEC que ponga un aspecto determinado en una señal, el FEC la pone en ese aspecto si la señal le responde y se lo comunica al IM. El FEC se descompone en tres subsistemas hardware (ver Figura 4.18): El EC (Element Controller) o módulo de lógica de elementos que es una configuración o bien dos de dos o dos de tres, según requiera el cliente y que es quien se comunica con el IM y el DS, el IC (Interface Controller), que es un módulo de interfaz también dos de dos y las tarjetas de interfaz con los elementos de campo, de las que hay dos tipos, las tarjetas de entrada-salida que disponen de 16 entradas y 16 salidas digitales y las tarjetas de control de señales, que disponen de 8 entradas y 8 salidas y que están especialmente preparadas para medir corriente y tensión en las lámparas, con el fin de detectar en las lámparas de las señales roturas y cortocircuitos. El FEC tiene también requisitos que afectan a la seguridad del sistema y por eso se ha incluido en la estrategia de validación. El FEC tiene una parte fija válida para todas las estaciones y otra parte configurable por medio de los datos de aplicación donde se definen los distintos elementos y su direccionamiento hardware.

- DS (Diagnostic System, Diagnosis). Guarda información de diagnosis de interés para la detección y corrección de errores y el mantenimiento. No siempre existe en todos los enclavamientos. No tiene redundancia de hardware. Por razones parecidas a las del OM y por el carácter no crítico de la funcionalidad de este subsistema, el DS se ha dejado fuera de la estrategia de validación.

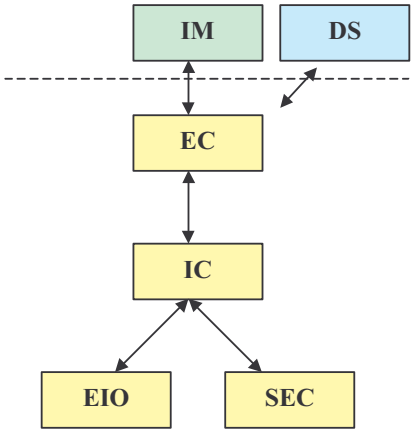


Figura 4.18 Diagrama del FEC

Como resumen, la estrategia de validación se ha aplicado al IM y al FEC, porque son las partes del enclavamiento responsables de los requisitos que afectan a la seguridad.

4.2.1.2 Las diferentes variantes

Los motivos por los que existen diferentes productos son las distintas configuraciones del hardware, las diferentes estaciones de ferrocarril y los distintos clientes (administraciones ferroviarias).

Producto = f (Configuración Hw, Estación, Cliente)

Las configuraciones hardware constan de varios procesadores trabajando en paralelo y sincronizándose para comparar los valores de sus entradas, sus salidas y parte de sus datos internos. En las configuraciones "dos de dos" hay dos procesadores funcionando juntos, que se paran cuando difieren en alguno de sus resultados. En una configuración "dos de tres", si un procesador está en desacuerdo con los otros dos, debe de pararse. La redundancia se aplica no sólo para los procesadores, sino para todos los elementos hardware del sistema, canales de comunicación entre los módulos y canales de entrada/salida a los elementos de campo. Esta redundancia hardware es obligatoria para cumplir con los requisitos de disponibilidad y de seguridad.

Puede elegirse entre distintos niveles de redundancia, lo que supone la existencia de diferentes componentes en la línea de productos. La configuración dos de tres da un mayor grado de disponibilidad [MeDu00]. Cuando existe como módulo hardware independiente, el IM es una configuración hardware dos de tres. El IM también puede ser una configuración dos de dos, integrado con el FEC en un mismo módulo hardware. El FEC, compuesto de EC e IC, es siempre una configuración dos de dos en el IC y en el EC puede ser un dos de dos, o un dos de tres según los requisitos de disponibilidad que haya en cada caso.

El sistema operativo utilizado en el IM y en el FEC, la plataforma de Alcatel TAS, es configurable para funcionar en modo dos de dos y dos de tres. La aplicación software (en este caso, la aplicación de enclavamiento electrónico) no se ve afectada. El código de aplicación sólo tiene llamadas a los servicios del sistema operativo, que siguen las directrices de la interfaz POSIX y la configuración que se requiera en cada caso se define aparte.

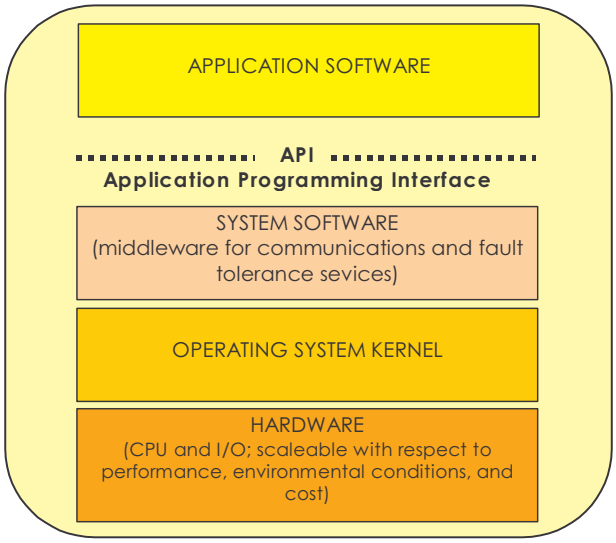


Figura 4.19 Estructura de la aplicación de enclavamiento

Las estaciones de ferrocarril son todas diferentes en cuanto a la topología y a los elementos hardware externos como agujas, señales, etc. Esta información se introduce en varios módulos de datos que se enlazan con el resto del software del sistema, que es válido para todas las estaciones, para formar el software específico de una cierta estación. Estos datos, llamados datos de aplicación, se generan con una herramienta software propietaria, la DPT ("Data Preparation Tool"). Esta herramienta da una interfaz más intuitiva al ingeniero de aplicación para generar los datos de la estación y comprueba la corrección de los datos.

Los datos de aplicación del IM incluyen fundamentalmente todos los elementos de campo que gobierna el IM correspondiente, los itinerarios posibles con los elementos de itinerario que le corresponden y las incompatibilidades que hay entre ellos. Los datos de aplicación del FEC indican los elementos de campo gobernados por el FEC correspondiente y la posición física de los contactos asignados a las entradas y las salidas de cada elemento (por ejemplo, en la entrada 30 y en la 31 el circuito de vía V23). En el caso especial de las señales, para cada señal se define en los datos de aplicación el comportamiento respecto a las fusiones de cada una de sus lámparas.

Otro conjunto de requisitos depende del entorno en el que funciona el sistema. Cada cliente (administración ferroviaria) tiene requisitos específicos, como por ejemplo, diferentes reglas de señalización, diferentes equipos externos, distintos procedimientos de trabajo, etc. Esto es especialmente notorio en el ámbito europeo, por motivos históricos [Bai95]. Las compañías que trabajan en más de un país, para optimizar los costes, deben de tener en cuenta esas diferencias, adaptando sus productos a los diferentes clientes, en vez de desarrollar de forma independiente en cada unidad nacional productos específicos para cada mercado [Val00], [Boe00].

En el caso del IM no se ha abordado de forma sistemática la tarea de hacer un IM genérico que se pueda adaptar para diferentes clientes. Los diferentes IM tiene la misma arquitectura software, pero la organización del desarrollo no está pensada para eso. En la actualidad, el IM se desarrolla en dos países, Alemania y España, para distintos clientes. Esto se debe a razones comerciales: el cliente de sistemas ferroviarios quiere un interlocutor único y de fácil acceso, y eso es difícil de hacer a distancia y mucho más para un producto que se hace a medida para cada cliente y a veces para una estación concreta. Se han articulado una serie de iniciativas de cooperación mutua, pero cada centro de desarrollo es responsable del software para sus clientes. Esto ha repercutido en la estrategia de validación que se ha orientado a probar los requisitos del cliente español (RENFE).

En el caso del FEC las diferencias de un cliente a otro vienen dadas o bien por los tipos de elemento hardware que se manejan o por los requisitos de seguridad que se impongan a la entrada/salida. Un ejemplo de la primera clase de variedad es RENFE en líneas convencionales tiene pasos a nivel, mientras que RENFE líneas de alta velocidad no los tiene. Un ejemplo de la segunda clase de variedad es la entrada salida básica, donde un contacto de entrada del campo se desdobla en dos entradas independientes para cada CPU del IC, o la entrada salida antivalente, donde desde el campo un mismo bit de información usa dos entradas, una en lógica positiva y la otra en lógica negativa, que a su vez se desdoblan también en dos entradas independientes para cada CPU del IC. Se ha conseguido estructurar el software definiendo el comportamiento de cada tipo de elemento de forma modular, y lo mismo con los tipos de entrada / salida. No es posible incluir en el ejecutable todas las clases de elemento y hacer una única versión para todos los clientes ya que dejar en el software código muerto disminuye la seguridad, y por ello hay que enlazar en el ejecutable del FEC sólo el código que se vaya a utilizar.

A continuación se describen las actividades de validación en el contexto de todo el proceso de desarrollo software.

4.2.2 Ciclo de vida software

La validación que se está describiendo se lleva a cabo en el marco de las normas CENELEC, que ya se han mencionado. Especialmente importante es la norma 50128 que define las diferentes etapas del ciclo de vida del software, y en función del nivel de seguridad que tiene que cumplir el sistema, propone una serie de técnicas y medidas para aplicar a cada fase, unas como obligatorias y otras como altamente recomendadas o simplemente recomendadas. Los enclavamientos tienen el nivel de seguridad 4, que es el más exigente (los SIL Safety Integrity Levels van de 0 a 4). En la figura puede verse el modelo de ciclo de vida según CENELEC, que no es sino un modelo de ciclo de vida en V detallado. Cada caja es una fase del ciclo de vida, y para acreditar el correcto cumplimiento de cada fase se necesitan unos determinados documentos. La certificación según CENELEC requiere el visto bueno de un auditor externo a la organización.

Para aplicar este modelo genérico de CENELEC dentro de la División de Automatización de Transporte de Alcatel se ha definido un ciclo de producto “Alcatel TAS Product Life Cycle” válido para todos los países con I+D propia (Alemania, Austria, Canadá, España, Francia y Portugal, eso sí, no todos tienen el mismo volumen). Eso permite una cierta homogeneidad en los procesos de desarrollo y facilita el intercambio de información entre los distintos países, aunque se necesita un cierto tiempo para generar toda la documentación requerida. Se establecen las siguientes fases en el desarrollo, siguiendo el modelo en V:

- Planificación: Incluye el Plan de Proyecto, Plan de calidad, Plan de Validación, Plan de gestión de Configuración, Plan de Verificación.
- Especificación de Requisitos de Sistema (SRS).
- Diseño de Arquitectura: genera el Documento de Diseño de Arquitectura (ADD).
- Diseño Detallado: genera el Documento de Diseño Detallado (DDD).
- Codificación: produce el Código Fuente.
- Pruebas Unitarias: Registradas en el Informe de pruebas Unitarias.
- Pruebas de Integración: Registradas en el Informe de Pruebas de Integración.
- Pruebas de Validación: Registradas en el Informe de Validación

Cada uno de los módulos del enclavamiento tiene todos estos documentos. Para la validación que aquí se explica son importantes los del IM y los del FEC.

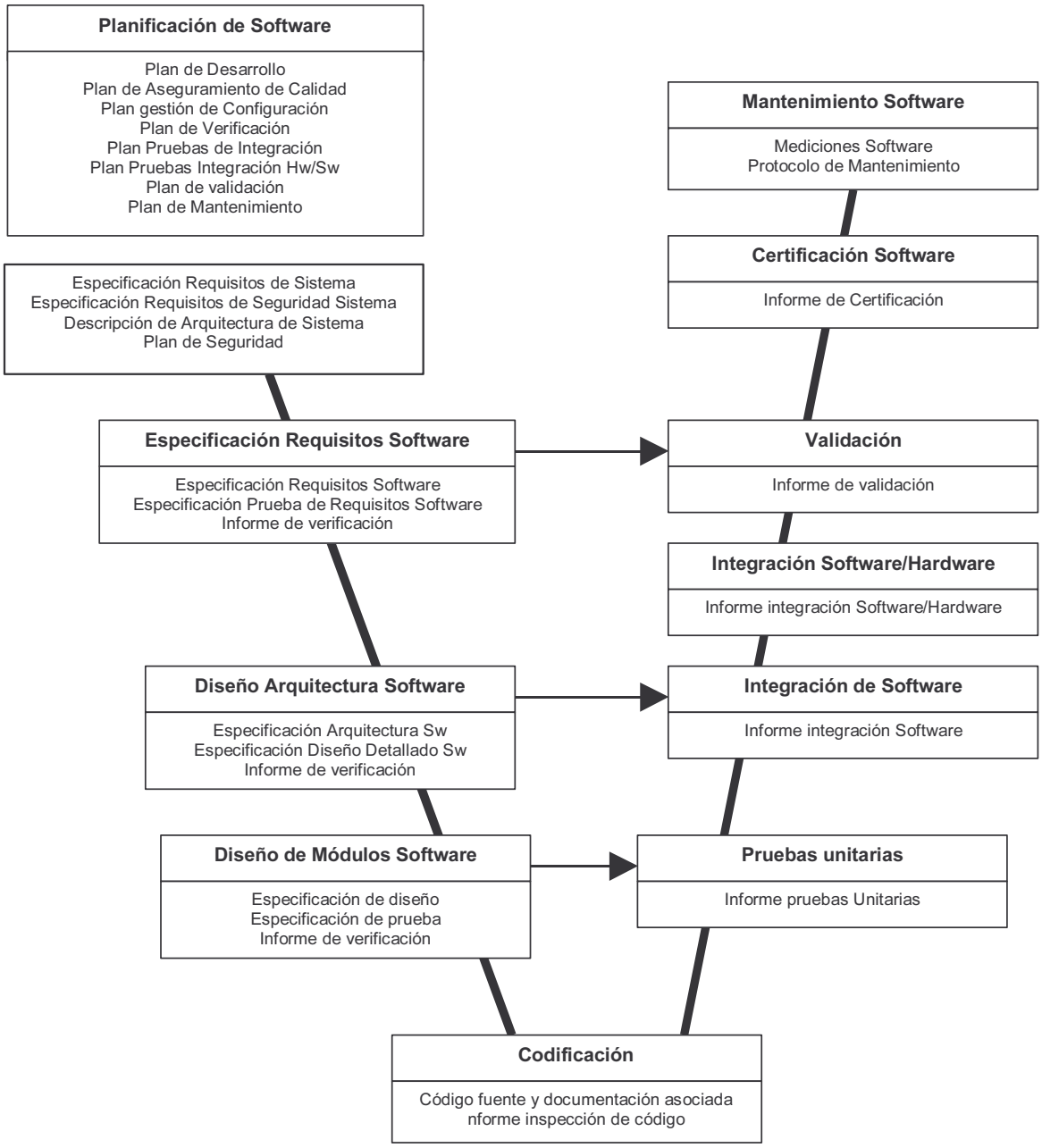


Figura 4.20 Ciclo de Vida según la norma CENELEC

Aunque estrictamente no forman parte del ciclo de vida del producto, en el caso de los enclavamientos electrónicos después de la validación hay unas pruebas de comprobación de seguridad que se realizan por parte de Alcatel. Por último, en la instalación final, el cliente hace las pruebas de aceptación.

Además, para cerrar cada fase tiene que existir una actividad de verificación de que se ha llevado a cabo correctamente. A estas verificaciones se les llama, en terminología ISO, revisiones del diseño. Alcatel Automatización de Transporte está certificada según la norma ISO 9000, con lo que ello implica: sistema de calidad, procedimientos, registros, auditorías internas y externas.

Para llevar cuenta de los defectos encontrados durante las distintas fases del ciclo de vida, especialmente en la validación, se utiliza una herramienta comercial de gestión de defectos. Los defectos se abren, se analizan y se cierran cuando se resuelven y todo queda registrado en la herramienta. Por un lado se tiene un mejor control de los defectos y se pueden sacar estadísticas de los fallos, pero por otro lado se pierde la agilidad de la comunicación oral.

Es fundamental para la validación acreditar documentalmente que todos los requisitos de sistema contenidos en la especificación se han probado. Esos registros son los que va a utilizar el auditor externo para certificar el sistema según CENELEC. Por eso es importante que exista trazabilidad de requisitos y casos de prueba.

4.2.3 La estrategia de validación en el ciclo de vida software

Como ya se ha dicho en la parte metodológica, la estrategia de validación es una extensión del ciclo de vida software en V. Se parte de los requisitos para especificar los casos de prueba, y a continuación para automatizar esos casos de prueba, hay una fase de diseño y otra de implementación.

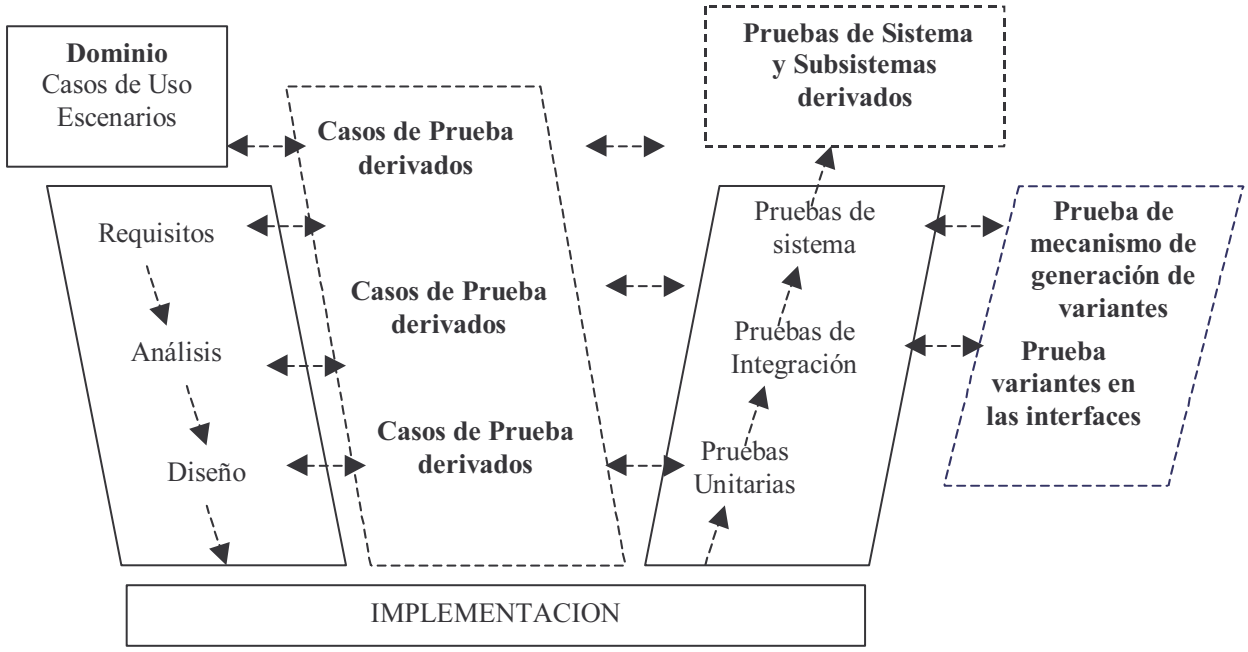


Figura 4.21 Modelo de ciclo de Vida en V extendido

4.2.3.1 Especificación de Clases de Prueba a partir de la especificación de Requisitos

La especificación de requisitos de sistema se ha hecho utilizando una herramienta comercial de gestión de requisitos, DOORS. La herramienta asigna a cada requisito un identificador y permite definir varios campos por requisito con el fin de clasificarlos. Los dos campos más importantes son el identificador de requisito y el texto. La herramienta permite, entre otras funciones, el control de usuarios, la generación de configuraciones de referencia ("baselines") y la creación de filtros para ver unos requisitos y ocultar otros. Desde la base de datos se pueden generar documentos de texto en los formatos de uso más habitual (HTML, Word, Excel, ASCII, etc.). La herramienta necesita un administrador central para administrar los usuarios y restaurar la base de datos en caso de problemas.

El idioma escogido para la especificación de requisitos ha sido el inglés. El motivo es que se ha hecho este documento no sólo pensando en el mercado español, sino en los mercados internacionales. Esto implica un esfuerzo adicional de traducción y, aunque hay diccionarios especializados en términos ferroviarios, no siempre se ha encontrado una traducción satisfactoria para cada término. Al igual que cada ferrocarril tiene funciones específicas, las palabras para designar esas funciones no tienen siempre fácil traducción.

Como se ha dicho antes, la variabilidad en esta línea de productos viene dada por las configuraciones de hardware, los datos de aplicación específicos de cada estación, y los diferentes clientes. En el nivel de Especificación de Requisitos, esta variabilidad se contempla de la siguiente forma:

Las variaciones en la configuración hardware elegida no afectan a los requisitos funcionales de la aplicación, ya que es la plataforma TAS quien se ocupa de gestionar las diferentes configuraciones. Para cada proyecto o instalación se especifica la configuración de hardware elegida, que tiene ya unos requisitos definidos, por ejemplo, una configuración dos de tres.

En cuanto a los datos de aplicación, en la especificación de requisitos se describe qué parámetros del enclavamiento se pueden definir con dichos datos y se hace una referencia a otros documentos en los que se describe de forma detallada el formato de los datos de aplicación.

La existencia de diferentes clientes se refleja definiendo un atributo para cada requisito que dice a que cliente corresponde, y en caso de corresponder a todos, se cataloga como general. Al elaborar por primera vez la Especificación de Requisitos, se contó con la colaboración de personal de Alcatel de otros países para identificar requisitos comunes a diferentes administraciones ferroviarias y catalogarlos como requisitos genéricos. La tarea resultó muy costosa, tanto desde el punto de vista técnico como organizativo. Se usaron como fuente de los requisitos lo que entonces había disponibles como requisitos del cliente en proyectos de cuatro países, uno de ellos no europeo.

4.2.3.2 El diseño de las pruebas

Diagramas de Estados

La especificación de requisitos es el punto de partida del desarrollo, pero también lo es para la validación, pues se valida contra los requisitos. Se han definido dentro de la especificación de requisitos clases, cada una de las cuales corresponde a un tipo de elemento de itinerario. Ejemplos de clases de prueba son el circuito de vía, la aguja, la señal, etc. En el caso de la señal, ha sido necesario hacer nuevas subdivisiones, pues el comportamiento cambia según el tipo de señal (señal de entrada, de avanzada, de maniobra, etc.). Como resultado del análisis de la especificación de requisitos, se definieron del orden de quince clases de prueba, algunas de las cuales corresponden a elementos que no estaban aún implementados en el sistema, pero que eran previsibles en proyectos futuros. Se han definido las mismas clases de prueba en el IM y en el FEC.

En cuanto a las subclases las del IM se han identificado: mandos, indicaciones, FAI y maniobra local. Estas dos últimas corresponden a funcionalidad que al comenzar la validación aún no se había implementado. La lección aprendida de esta experiencia es que no merece la pena ese esfuerzo de preparar la validación de funciones aún no implementadas.

Es en el nivel de subclase donde se han realizado los diagramas de estados anotados como el de la figura que se explican en la parte metodológica de la Tesis Doctoral.

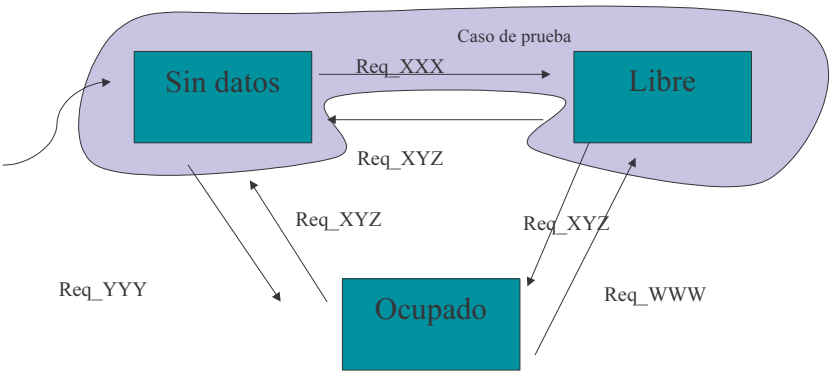


Figura 4.22 Ejemplo de diagrama de transición de estados

En el caso del IM, el comportamiento de cada elemento de itinerario depende no sólo del mismo sino del estado del itinerario, que a su vez es función del estado de todos los elementos que lo componen. Un circuito de vía puede estar libre sin más para el FEC, pero para el IM es distinto si es libre dentro de un itinerario ya establecido o libre dentro de un itinerario disuelto por emergencia o libre en un itinerario disuelto por paso de tren, etc. Modelar todo este conjunto de condiciones simultáneamente en una máquina de estados es complejo aun utilizando jerarquías de estados, y lo que se hizo fue construir diagramas de estados parciales que se incluyeron en una tabla de estados general, en la fase de diseño de las pruebas. Se hizo la validación del IM sólo en el caso de la señalización de RENFE. En este caso, existen tres tipos de itinerarios: itinerarios normales, maniobras y rebases autorizados (un tipo de itinerario especial para moverse a poca velocidad en la estación y que no tiene deslizamiento). Simplemente por esto, el número de pruebas ya se triplica. La tabla que hay a continuación es un ejemplo real (por eso está en inglés, ya que se escogió hacer así para poder usar los documentos de validación para certificaciones internacionales) de un caso de prueba sencillo.

Test Case Identifier:	TCL-1.1-TC-2
Diagram:	Track section states.
Tittle:	Failed to clear
Test Case Description:	The track section state is "Failed". There is a notification that the track station state is "Clear" and the track section state becomes "Clear". The new state is indicated to the MMI.
Test Class Dependency:	TCL-1
Requirements Identifiers:	rim_58; rim_92; rim_56; rim_5646; rim_5234
Input Specification:	The Track section state is Failed. (The other track section states are irrelevant).
Output Specifications:	The track station state is Clear. The rest of the states are not modified. The new state "Clear" is indicated to the MMI.
Event:	The notification that the track section state is "Clear" (There is no train in the section).
Environmental Needs:	Every element in the station is "Clear" except this track section that its state is failed. There is no route, no block, no Local Operation established, no ARS activated and the station is under local control.

Tabla 4.1 Ejemplo de Caso de Prueba para el módulo IM

Cada caso de prueba está definido por los siguientes parámetros:

- Identificador de caso de prueba: Permite saber para cada "campaña" de validación si se ha ejecutado o no.
- Titulo del caso de prueba.

- Descripción de la prueba: Sirve como aclaración en caso de que el diagrama de estados no sea suficiente.
- Diagramas de estados asociado al caso de prueba.
- Clase de prueba a la que pertenece el caso de prueba.
- Identificadores de los requisitos asociados a esta transición: Se determinan mediante análisis de la especificación de requisitos.
- Entrada: Es el estado inicial.
- Salida: Es el estado final.
- Evento: Produce el paso del estado inicial al estado final.
- Condiciones del entorno: Pueden llegar a ser muy complejas en el caso del IM.

En el caso del FEC se han definido sólo las subclases de mandos y la de indicaciones. En este caso, sí que se cubre con diagramas de estados la funcionalidad completa del subsistema, y la estrategia de validación se ha demostrado especialmente adecuada. Los diagramas de estados de los elementos de ruta son lo suficientemente sencillos como para permitir la representación gráfica y, salvo excepciones, el estado de un elemento no depende del estado de otros elementos del sistema. Esto se debe a que la lógica del FEC es mucho menos compleja, ya que no es el responsable del tratamiento de itinerarios, sino únicamente de controlar el estado de los elementos exteriores bajo las órdenes del IM. Este proceso se ha realizado sin ninguna herramienta de soporte, y es aquí donde se ha visto el interés de la aplicación informática que se describe en la parte práctica de la Tesis Doctoral.

En la figura se puede ver un ejemplo de diagrama de estados (en la parte izquierda) correspondiente a una clase de prueba sencilla, el circuito de vía, y cómo se han derivado a partir de él los casos de prueba (en la parte derecha).

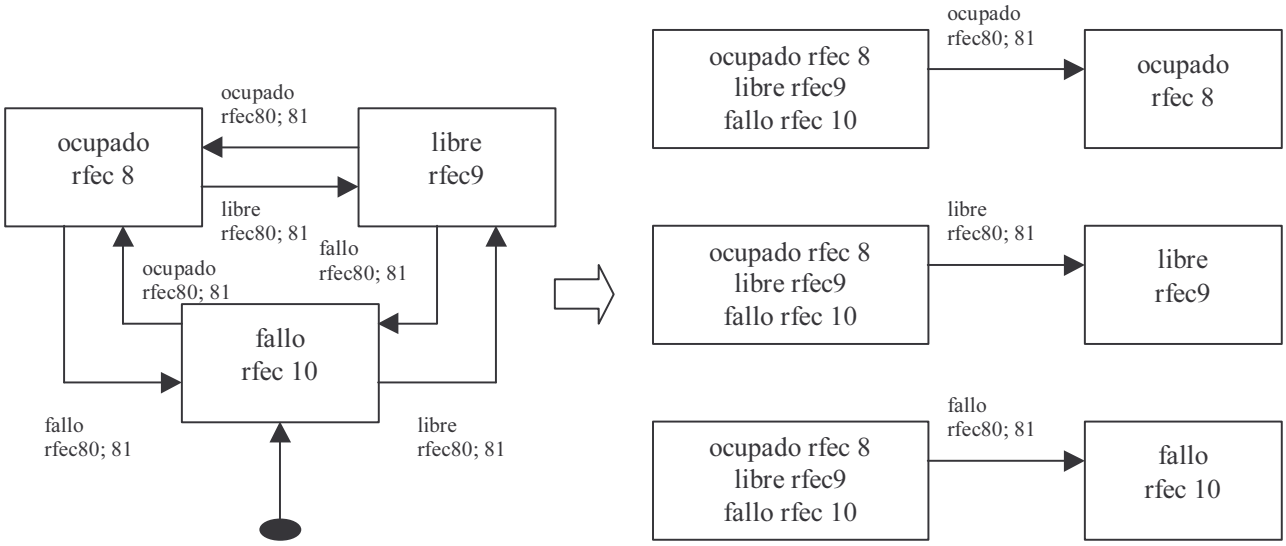


Figura 4.23 Obtención de casos de prueba a partir del diagrama de estados de la clase de prueba

El conjunto de todos los casos de prueba se denomina Especificación de Casos de Prueba o Test Case Specification.

Escenarios

Las subclases modeladas con diagramas de estados no cubren toda la funcionalidad del sistema en el caso del IM, puesto que este módulo controla además de los elementos de itinerario por separado, situaciones más complejas en las que intervienen elementos de tipos diferentes. Esta carencia fue detectada durante el diseño de la estrategia de validación y se optó por poner estos escenarios complejos por escrito y probarlos manualmente, ya que hacerlo de forma automática requiere un gran esfuerzo.

La consecuencia de ello es que esta parte de la validación requiere más tiempo y es más problemática porque toda la responsabilidad recae sobre el validador, que tiene que decidir, con la información del protocolo de pruebas y su experiencia, si la prueba ha resultado correcta o no. Desde este punto de vista, la estrategia de validación está en desventaja frente a las pruebas de caja negra que se hacen para la comprobación de seguridad. El motivo es que los escenarios de validación siempre tienen un carácter genérico, para poder aplicarlos sobre el máximo número de configuraciones, mientras que en las pruebas de comprobación de seguridad se realizan a medida para cada instalación.

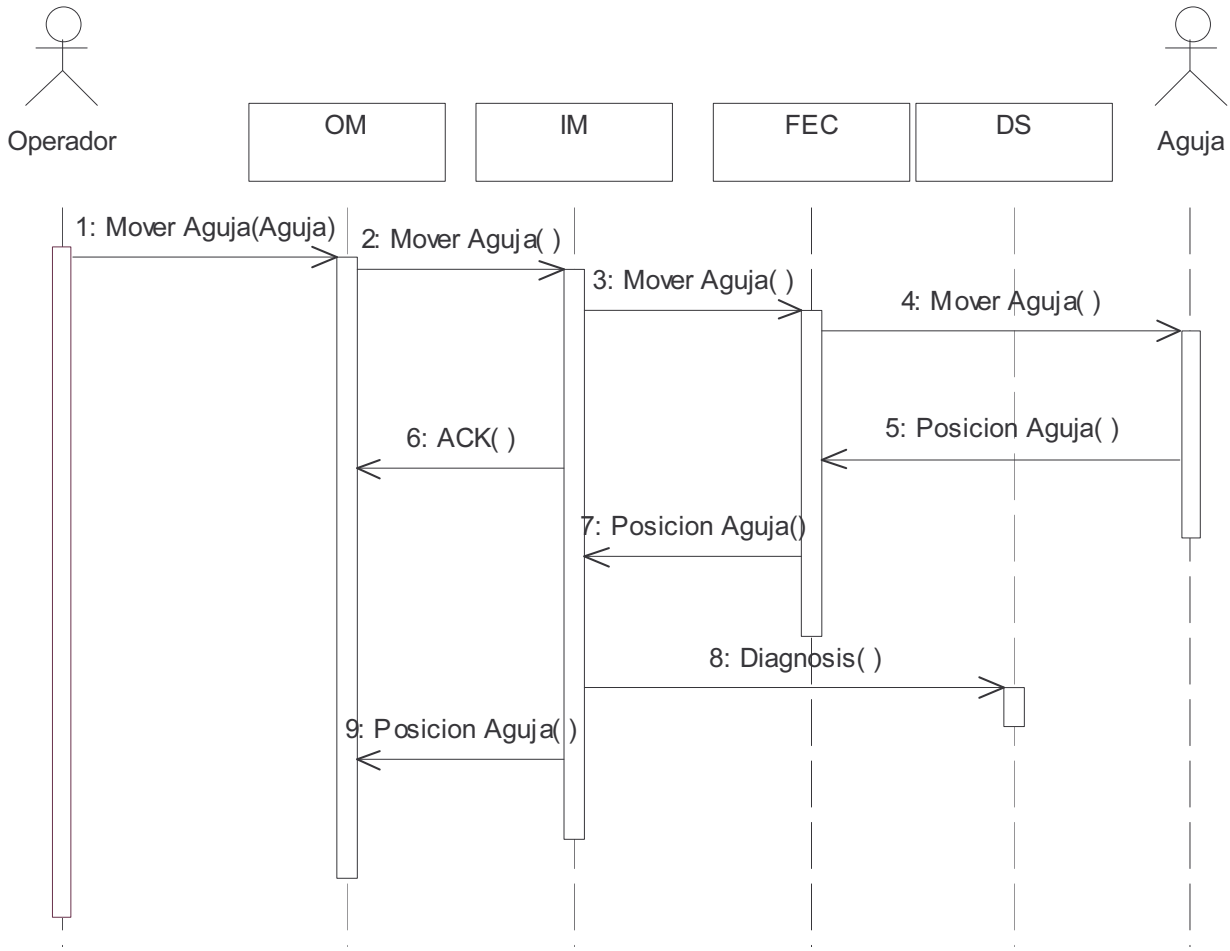


Figura 4.24 Ejemplo de escenario

Aparte de eso, los escenarios tienen el problema de que a medida que son más complejos, pueden tener diferentes ejecuciones. Un recorrido sistemático de todas las ejecuciones alternativas es un esfuerzo muy costoso si se hace exclusivamente de forma manual.

4.2.3.3 Descripción general del entorno de validación para los dos casos de estudio

En la Figura 4.17 se muestra la estructura del sistema completo y en la Figura 4.18 la estructura del FEC. En las figuras que hay a continuación se representan las partes del sistema que se simulan por parte del entorno de validación.

El entorno de validación del IM simula, como aparece en la figura, todos los módulos del sistema con los que se comunica el IM: el OM, el FEC y el DS. El entorno manda al IM los mismos mensajes que le mandarían en funcionamiento real dichos módulos y recibe las respuestas del IM. También puede acceder a ciertas variables de estado interno del IM, tanto para leer su valor como para modificarlo. El entorno se ejecuta en la misma plataforma hardware que el IM y se ha diseñado para que ejecute en secuencia todos los casos que haya establecido el validador en un script de prueba que se enlaza con el IM antes de ejecutarlo. Los mensajes entre IM y OM y entre FEC e IM se guardan en un fichero de registro. Los mensajes al DS se pueden guardar o no según se necesite. Los módulos con los que se comunica el IM, a saber, el FEC, DS y OM se simulan mediante *"dummies"*, es decir módulos software sin funcionalidad pero que implementan las interfaces correspondientes.

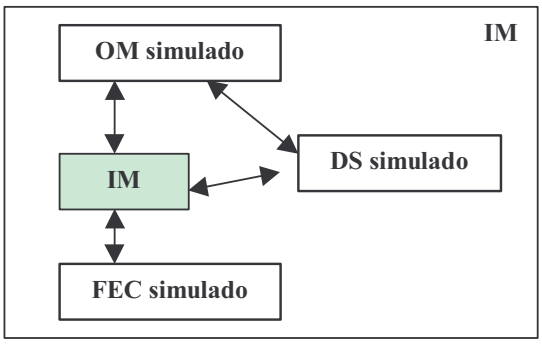


Figura 4.25 Entorno de validación para el IM

El entorno de validación del FEC simula únicamente el DS y el IM, pues son los módulos del sistema que se comunican con el FEC. Los elementos de hardware externos no se simulan sino que se conectan de la misma manera que si se conectaran a un enclavamiento real. Aquí hay que aclarar que los elementos de hardware del enclavamiento tienen una configuración simplificada que se usa en las pruebas para hacerlas más fáciles. Por ejemplo, es posible probar las agujas, mediante un sencillo hardware auxiliar sin conectarles sus motores.

Aun teniendo la configuración de elementos de hardware más simple posible, simular su comportamiento no es trivial. Por eso en una primera fase se decidió que el entorno de validación no simulara el funcionamiento del hardware sino que fuera el operador el que actuara sobre dicho hardware o verificara su estado. Por ejemplo, ver que una cierta lámpara se ha encendido o apagado, ver que unos relés han cambiado su posición, etc. El entorno puede también, haciendo el papel del IM, solicitar al FEC que actúe sobre un elemento, por ejemplo, que cambie el aspecto de una señal, y verificar a continuación que el FEC ha obedecido su orden, leyendo el estado de la señal y analizando la notificación del FEC tras realizar el mando.

El entorno del FEC se ha diseñado para que sea semi-interactivo debido a la necesidad de que el operador confirme el estado inicial o el final de los casos de prueba. El validador comunica al FEC su

confirmación a través de un PC. Los mensajes entre el IM y el FEC y entre el EC y el IC se guardan en un fichero de registro. Los mensajes al DS se pueden guardar o no según se necesite.

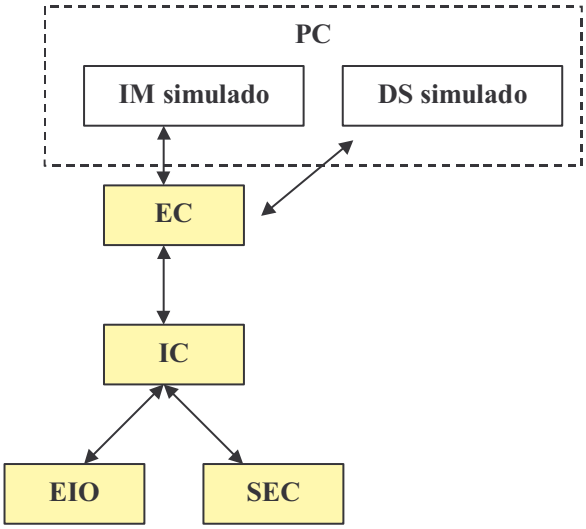


Figura 4.26 Entorno de validación del FEC

4.2.3.4 El diseño detallado de las pruebas

Cada caso de prueba especificado en la fase anterior se implementa ahora para poder ejecutarse de forma automatizada. La estructuración de los casos de prueba en clases se sigue manteniendo. Todos los casos de prueba de una clase se implementan en una misma hoja de cálculo, a partir de la cual se generan unos ficheros fuente en lenguaje C que se enlazan con el software del módulo que se quiere validar (el FEC o el IM). La hoja de cálculo contiene la información básica de la especificación del caso de prueba y se denomina especificación de datos de prueba:

Caso de Prueba = F (ID, Entrada, Evento, Salida)

- Identificador de caso de prueba: Permite saber para cada "campaña" de validación si se ha ejecutado o no.
- Entrada: Es el estado inicial.
- Evento: Produce el paso del estado inicial al estado final.
- Salida: Es el estado final.

En la Tabla 4.2 puede verse una parte de la especificación del caso de prueba del circuito de vía para el FEC, en uno de los casos de prueba (el TI_2). En este caso, se representa la situación de recibir una indicación de que el circuito de vía está ocupado y el estado del circuito de vía cambia a libre.

Type	***ID***	***Information***	***Description***	***TI1_2***
S	1	***Indications***		TI_OCUP_FIELD_MASK
I	1	***Transitions***	Track Occupied	
I	2		Track Clear	TI_FREE_FIELD_MASK
I	3		No Data	

Tabla 4.2 Ejemplo de Caso de Prueba del módulo FEC

En el entorno del FEC para cada caso de prueba se definen los datos siguientes:

- Identificador de caso de prueba: Permite saber para cada "campaña" de validación si se ha ejecutado o no.

- Entrada: Es el estado inicial del elemento que se está probando. El entorno de validación en el caso del FEC es semiautomático y requiere que el validador ponga el elemento físico en el estado inicial que el entorno le solicita. Por ejemplo, en el caso del ejemplo, debe de actuar sobre el circuito de vía para ponerlo en estado ocupado. En el caso del circuito de vía, lo hace manipulando unos interruptores que están conectados a las dos entradas de la tarjeta de entrada/salida del enclavamiento que corresponden a un cierto circuito de vía. El entorno de validación dispone de un mecanismo de lectura del estado del elemento que se prueba. En el caso del ejemplo, comprobaría que el validador ha puesto el circuito de vía realmente en estado ocupado y le pediría entonces su visto bueno para comenzar la prueba.

- Evento: Produce el paso del estado inicial al estado final. Puede ser o bien una acción que el operador debe realizar manualmente o una acción que el entorno efectúa de forma automática, según el caso de prueba. En el caso del ejemplo, es una acción del operador, que debe de cambiar las posiciones de los interruptores del circuito de vía para que su estado sea libre.

- Salida: Es el estado final. El entorno simula el IM frente al FEC y es capaz de interceptar los mensajes que le manda el FEC y analizarlos. Si coincide con la información que tiene de la prueba, da el caso por pasado, y lo refleja en las estadísticas de la "campaña" de validación actual, y en caso contrario, da el caso por fallido. Volviendo al ejemplo, si todo ha funcionado correctamente, el FEC reconoce el cambio en el estado de la entrada/salida y lo ha notifica al IM mediante un mensaje, que en vez de ser recibido por un IM real, es recibido por el entorno y comparado con el mensaje correcto que corresponde al caso de prueba.

Para dar una idea de la diferencia de complejidad entre la lógica del FEC y la del IM, se muestra el mismo caso de prueba (el TC1) para el IM en la Tabla 4.3. Lo que quiere decir es que si se produce el evento "Train_leaves", debe cambiar el estado de los indicadores 1, 3 y 4, que inicialmente están a FALSE, TRUE y TRUE.

Para hacer más ágiles las pruebas, se utiliza una hoja de cálculo separada para manejar la información de trazabilidad de requisitos, tanto en el IM como en el FEC. Esto ha sido así por razones de tipo práctico: hay muchas más campañas de validación (pruebas de validación para un proyecto real), que certificaciones, que son las que exigen documentar la trazabilidad. No se ha utilizado una herramienta de soporte para esta tarea, y realizarlo manualmente es bastante laborioso. Por eso, en las herramientas propuestas en la parte práctica de la Tesis Doctoral se aborda también este problema.

Se adoptó una solución de compromiso entre validar un software idéntico al que se va a poner en servicio y la necesidad de poder monitorizar el módulo validado para automatizar las pruebas. Las pruebas de comprobación de seguridad se siguen realizando y esas pruebas sí que utilizan la misma versión de software que funciona en la instalación real. Pero el objetivo de la estrategia de validación expuesta es algo diferente. Es importante comprobar los requisitos que afectan a la seguridad, pero también se quiere cumplir con las exigencias de los estándares vigentes (CENELEC):

Type	*ID*	*Information*	*Description*	*TC1*
S	1	OCC-PH	Clear	0
S	2		Occupied	0
S	3		Failed	1
S	4	OCC-LOG	Log.Occupied	1
S	5		FreeAsOccupied	0
S	6		NotSEQOccupOrFreed	0
S	7		UnexpOcc	0
S	8		SeqOcc	0
S	9		BSEQ	0
S	10	INH	R.Inhibition	0
NI:S	11		R.Inhibition by Destination	0
S	13		MRPmr	0
S	14		MOmr	0
NI:S	15		MAmr	0
S	16		MFPmr	0
S	17	BL	MB	0
NI:S	18	LO	MLO	
E	1	***Events***	Train_Occupies	
E	2		Train_Leaves	1,3,4
E	3		No_Data	

Tabla 4.3 Ejemplo de Caso de Prueba del módulo IM

- Medir cobertura de requisitos.
- Medir cobertura de código ejecutado en las pruebas.

El primer objetivo se ha logrado mediante el método que se ha expuesto, pues cada caso de prueba tiene unos requisitos asociados. Sabiendo qué pruebas se han realizado, se sabe qué requisitos se han verificado. Si queda alguno sin probar, se diseñan nuevos casos de prueba que en el peor de los casos se ejecutan de forma manual sin ningún tipo de soporte automático.

El segundo objetivo se ha conseguido utilizando una herramienta de medida de cobertura disponible, tanto en el IM como en el FEC que se llama gcov y que viene incorporada en el compilador de C de GNU. Para medir cobertura con gcov, basta compilar la aplicación correspondiente activando dos opciones de compilación adicionales y tras enlazar, se ejecuta la aplicación y cuando la ejecución termina, gcov ha generado un fichero de registro que contiene las líneas de código que se han ejecutado. A partir de este fichero mediante comandos estándar de UNIX se genera otro nuevo fichero de texto ASCII que se puede incorporar al informe de validación correspondiente. Los porcentajes de medición de cobertura usando este método no son demasiado elevados, puesto en los casos de prueba de la validación no se ejecuta todo el tratamiento de excepciones software que existe en el código debido a las exigencias de los estándares de codificación, basados en [EBA94] y [EBA99].

Por esta razón se ha abierto una nueva línea de trabajo para la medida de cobertura que está basada en pruebas unitarias. Las pruebas unitarias se ejecutan sobre PC utilizando una herramienta comercial, que en una primera fase ha sido Insure de la casa Parasoft. Aquí se han aprovechado las ventajas de portabilidad que tiene el lenguaje C para ser compilado en diferentes plataformas hardware. Con este método se obtiene mayores porcentajes, pero con el esfuerzo añadido de tener que escribir *stubs* (programas auxiliares) de prueba para cada función software que se quiere verificar.

No se ha llevado a cabo por falta de tiempo hasta la fecha una validación según CENELEC de la estrategia de validación y de sus mecanismos. En la práctica, al comenzar una "campaña" de

validación hay una primera fase de "take-over" o toma de contacto en la que se ejecutan unos casos de prueba básicos que tienen que funcionar siempre y si esos no pasan, se empieza a mirar hacia atrás para localizar el problema, ya sea en el software que se está validando o en el entorno de validación. Si el problema está en el software que se valida, se interrumpe la validación y, de acuerdo con el equipo de desarrollo, se espera obtener una nueva versión corregida. De esta forma se evita pasar las pruebas de validación sobre una versión de software que va a ser corregida de todas las maneras.

4.2.3.5 Implementación de las pruebas

Se entiende aquí por implementación de las pruebas la generación de scripts de prueba a partir de la especificación que se ha realizado en la fase de diseño detallado de pruebas. Los scripts de prueba son ficheros escritos en lenguaje C que se compilan conjuntamente con el sistema software que se quiere validar. Los scripts se generan a partir de las hojas de cálculo que contienen la especificación de diseño detallado. En la Figura 4.27 puede verse una descripción del proceso en general, válida tanto para el IM como para el FEC.

Existe primeramente una fase de preparación de las pruebas en las hojas de cálculo, definiendo para cada prueba sus cuatro parámetros:

Caso de Prueba = F(ID, Entrada, Evento, Salida)

Eso se transforma, por medio de una herramienta software propietaria, en ficheros de lenguaje C para poderlo enlazar con el software del módulo que se quiere validar (IM o FEC). Aquí existe ya la posibilidad de según los ficheros que se escojan para enlazar, el realizar unas pruebas u otras. Esta flexibilidad del entorno es útil para gestionar las diferentes variaciones en el software del IM y del FEC debidas a los diferentes clientes y diferentes estaciones.

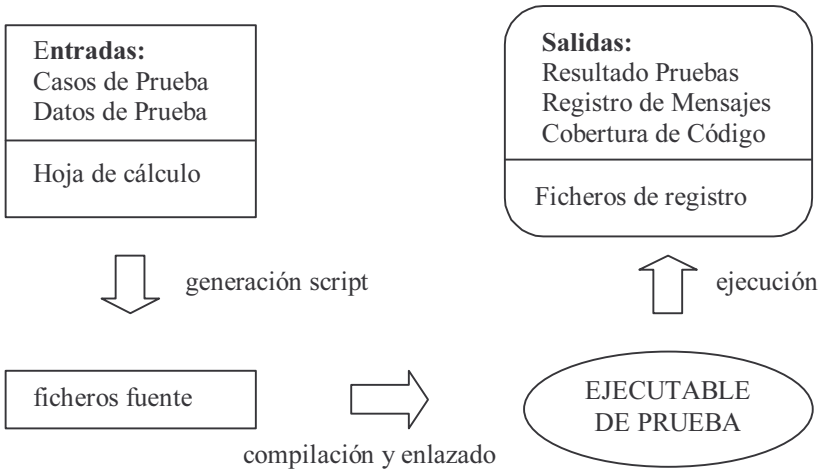


Figura 4.27 Esquema del Proceso de Automatización

4.2.3.6 La ejecución automatizada de las pruebas

Validación del IM

Los validadores deciden a la hora de construir el ejecutable qué casos de prueba se quieren pasar, generan los scripts de prueba correspondientes a esos casos de prueba y los incluyen en el ejecutable. Por ejemplo, si se quieren probar los casos de prueba del circuito de vía, se selecciona la hoja de cálculo donde están especificados y se manda generar los scripts de prueba que se enlazan con el IM. Hay información que debe de modificarse según el contenido de los datos de aplicación del IM. Es decir, que los datos de prueba no son iguales para todas las estaciones. Una posible mejora es que el entorno realice esta transformación de forma automática.

El operador arranca el IM a través de la consola del sistema y los casos de prueba se ejecutan uno detrás de otro. Lógicamente, después de pasar una prueba, se tiene que "limpiar" los estados internos de forma que no se influya en la siguiente. La ejecución de estos casos de prueba genera un archivo de registro de las salidas del sistema y un informe de cobertura de las pruebas. El registro de las salidas del sistema contiene las salidas de los casos de prueba y los mensajes que ha mandado el IM como consecuencia de los eventos generados por el entorno de validación. Este registro se analiza después de la ejecución con una herramienta propietaria, que determina para caso de prueba si ha producido la salida esperada o no, y da el caso como pasado o como fallido, generándose un fichero de resultados (ver los ejemplos del fichero de registro adjuntos). La cobertura de código (qué sentencias del IM se han ejecutado con esas pruebas) se mide, como se ha dicho ya, con una herramienta que viene incluida en el entorno de desarrollo del IM (entorno GNU). Los defectos encontrados se introducen en una herramienta comercial de gestión de defectos y en el informe de validación correspondiente.

En el caso del IM se aplicó esta estrategia únicamente para el IM específico de RENFE, con lo cual la variabilidad de los productos está restringida sólo a los diferentes datos de aplicación para cada estación.

```
Test Case Case s1_0
TS_IsFree(21): 0 0
TS_IsOcuppiedPhysically(21): 0 1
TS_IsFailed(21): 1 0
TS_IsBlocked(21): 1 1
TS_IsInMainRoute(21): 1 1
TS_IsInOverlapOfMainRoute(21): 1 1
TS_IsInAproximationRoute(21): 1 1
TS_IsInFlankProtectionRoute(21): 1 1
TS_IsInBlockRoute(21): 1 1


Case s1_0 Init OK
Queue 6, n-mssg 0
T: 961085411,764 -- pos 6 s,1,0,1,0,15,0,0,0,3,0,0,0,
T: 961085411,865 -- pos 6 r,1,0,1,0,15,0,0,0,3,0,0,0,
...
Queue 0, n-mssg 168
T: 961085411,967 -- pos 0 s,4a,15,0,1,87,1,0,
Queue 4, n-mssg 0
T: 961085411,968 -- pos 4 s,3,0,1,0,23,0,1,0,0,0,0,0,0,0,
...
Queue 0, n-mssg 169
T: 961085412,475 -- pos 0 s,4b,3,1,40,0,0,0,56,0,0,0,1,1,
Queue 0, n-mssg 170
T: 961085412,476 -- pos 0 s,4b,3,1,40,0,0,0,55,0,0,0,1,1,
...
Queue 0, n-mssg 171
T: 961085412,578 -- pos 0 s,4a,f,0,1,3,0,0,
```

Figura 4.28 Ejemplo de registro de salidas del entorno del IM

```
Test Case Case s1_0: OK
Test Case Case s1_1: OK
Test Case Case s1_2: OK
Test Case Case s1_3: NOK -->
TS_IsInMainRoute(21): 1 0
Test Case Case s1_4: OK
Test Case Case s1_5: OK
Test Case Case s1_6: OK
Test Case Case s1_7: OK
Test Case Case s1_8: OK
Test Case Case s1_9: OK
Test Case Case s1_10: OK
Test Case Case s1_11: OK
Test Case Case s1_12: OK
Test Case Case s1_13: OK
Test Case Case s1_15: OK
Test Case Case s1_15: OK
Test Case Case s1_16: OK
Test Case Case s1_17: OK
Test Case Case s1_19: OK
Test Case Case s1_20: OK
```

Figura 4.29 Ejemplo de fichero de resultados del entorno del IM

Validación del FEC

El entorno de validación del FEC es, como ya se ha dicho, diferente, pues este subsistema controla elementos de hardware como agujas, señales y circuitos de vía y es necesario durante la validación el supervisar las reacciones de estos elementos físicos. Como una simulación fidedigna realizada exclusivamente mediante software de estos elementos es costosa de conseguir, se decidió que el entorno del FEC fuera interactivo, a diferencia del IM. Los validadores "dialogan" con el FEC a través de un PC, seleccionan el caso de prueba que quieren ejecutar y el entorno, según la prueba de la que se trate, realiza un mando sobre el FEC o solicita al validador que actúe sobre el equipo exterior para que tenga el estado inicial requerido por la prueba.

```
Test Case SG1_19
EC-IC: 42001000100421010001004220100010042301001111 9
State: 44
State: 44
State: 44
State: 44
State: 44
State: 44
State: 44
State: 44
State: 44
State: 44
State: 44
State: 44
State: 44
State: 44
EC-IM: 3411080000FF0000
EC-IM: 34114020800000
EC-IM: 3411080000FF0000
EC-IM: 34114020800000
EC-IM: 3411070001400000
EC-IC: 42001000100421010001004220100010042300000000 4111109
State: FF

The Test Case is Passed
```

Figura 4.30 Ejemplo de fichero de registro del entorno del FEC

El entorno, si es necesario, solicita al validador que confirme si el equipo exterior ha reaccionado de acuerdo a lo especificado en la prueba. El entorno registra los mensajes y estados internos relevantes

para el caso de prueba, y al igual que en el entorno del IM, un registro de cobertura de código en un fichero aparte. En la Figura 4.30 se muestra un ejemplo del registro de un caso de prueba, que contiene la clase de prueba a la que pertenece (en el ejemplo es la clase SG, correspondiente a las señales), el estado del elemento durante la prueba, el registro de los mensajes intercambiados en el sistema durante la ejecución de la prueba y el resultado (fallido, pasado).

En el FEC los casos de prueba se clasifican en una primera división según el cliente. Las clases de prueba están separadas para cada cliente, y con la información de los ficheros de registro puede medirse en cada momento el progreso de la validación para un cliente dado. Es posible validar en paralelo diferentes clientes con una medición exacta del estado de la validación para cada uno de ellos. Aunque hay requisitos genéricos para todos los clientes, no se ha implementado una jerarquía en los casos de prueba como la que se propone en [Gre00]. Existen colecciones de casos de prueba para cada cliente que tienen la misma estructura y que se producen con el mismo método, ejecutándose en el mismo entorno y eso permite una cierta reutilización de las pruebas, pero lo habitual es que para cada cliente haya que realizar todas, por motivos de seguridad.

4.2.3.7 La estrategia de validación aplicada a los diferentes miembros de la línea de productos

Como se ha dicho hasta ahora, se han identificado tres fuentes de variación en la línea de productos de sistemas de control del tráfico ferroviario: diferentes plataformas hardware, diferentes estaciones de ferrocarril y diferentes clientes.

Producto = f (Configuración Hw, Estación, Cliente)

Se han implementado diferentes soluciones para tener en cuenta esas variaciones durante las pruebas:

Las variaciones debidas a las distintas configuraciones hardware se prueban manualmente usando un protocolo de pruebas. El protocolo contiene una relación de casos de prueba que deben de ejecutarse, una breve descripción de cada caso de prueba y su resultado esperado. El sistema operativo [MeDu00] es adaptable a las diferentes configuraciones de forma que el software de aplicación no debe de modificarse. Estas variaciones no afectan al entorno de pruebas automáticas, puesto que el entorno esta enfocado a probar el software de aplicación.

El entorno de pruebas automatizadas soporta diferentes estaciones. Los datos de prueba se preparan manualmente de acuerdo a los datos de la configuración particular (estación) sobre la que se desean ejecutar las pruebas.

La variación más importante y la más difícil de manejar es la debida a la existencia de diferentes clientes con requisitos dispares y a menudo incompatibles.

La documentación de diseño se genera para cada cliente por separado. Si las diferencias entre el diseño requerido para un nuevo cliente y otro cliente ya existente no son significativas, los documentos de diseño del nuevo cliente contienen una referencia a los documentos del otro cliente y describen únicamente las diferencias que hay entre los dos. La misma estrategia se ha utilizado hasta ahora con los escenarios de prueba, pues son también documentos que se usan para las pruebas aparte de para el diseño.

Los cambios de los requisitos debidos a los diferentes clientes influyen en el entorno de pruebas pues pueden implicar un cambio en los diagramas de transición de estados o en los escenarios. Aunque los clientes tienen en común los llamados requisitos genéricos, se han definido en un primer paso clases

de prueba separadas para cada cliente, y por tanto, diagramas de estados y escenarios separados, pues eso simplifica la gestión de configuración de los artefactos de prueba.

Es posible conocer la cobertura de requisitos alcanzada a partir de los casos de prueba que se hayan realizado para cada cliente, ya que la estrategia de validación permite trazar los casos de prueba respecto de los requisitos y viceversa. Si el nivel de cobertura alcanzado no es el planificado, se añaden nuevos casos de prueba para conseguirlo.

Los cambios en los requisitos debido a los distintos clientes pueden afectar al entorno de pruebas de formas diferentes. Hay requisitos que afectan en conjunto al enclavamiento, como por ejemplo, que el tiempo que debe de tardar el enclavamiento en indicar al operador que un circuito de vía se ha ocupado no debe de exceder los 500 milisegundos. Otros requisitos están claramente asociados a un módulo concreto, como por ejemplo, el EC es responsable de activar el motor una aguja mientras que la aguja no alcance su nueva posición, hasta un máximo de diez segundos.

Teóricamente, si se prueba con éxito un requisito genérico para un cierto producto, puede darse por verificado para todos los demás productos. No se está aplicando esta política de forma sistemática, pues los fallos en este dominio de aplicación son muy críticos: existe un conjunto mínimo de pruebas de requisitos genéricos que hay que hacer absolutamente a todos los productos y que se hace de forma manual. También se prefiere repetir las pruebas de requisitos genéricos automatizadas sobre todos los productos que evitarlas, para mayor seguridad y porque los registros que se generan durante las pruebas automáticas son más detallados que los de las pruebas que se hacen de forma manual.

4.2.4 Resultados alcanzados

El desarrollo por medio de Líneas de Producto Software por parte de Alcatel TAS ha obtenido logros muy significativos. Por un lado, la plataforma, solamente en lo que se refiere a enclavamientos electrónicos, ha unificado en dos plataformas hardware comerciales y un sólo sistema operativo desarrollado por Alcatel sólo parcialmente lo que en la generación de sistemas previa necesitaba cuatro plataformas hardware diferentes propietarias y tres sistemas operativos también propietarios, con la consiguiente reducción de coste a nivel de toda la compañía. La gestión conjunta de los requisitos de los diferentes clientes ha tenido como fruto una reducción importante en el esfuerzo de desarrollo. Un ejemplo muy claro es el caso del tratamiento de las señales en el FEC, donde una única aplicación software genérica es capaz de hacer lo que la generación previa de sistemas necesitaba una aplicación específica por cada cliente, habiendo logrado una reducción estimada del esfuerzo de desarrollo en este punto superior al ochenta por ciento.

El entorno de validación de Alcatel TAS realiza pruebas de forma automática y genera, aparte del registro de resultados de las pruebas, un registro de medida de cobertura y otro registro de cobertura de requisitos. La primera versión del entorno del IM comprende en torno a mil casos de prueba para un único cliente, y el entorno del FEC en torno a 1400 para dos clientes distintos. El entorno de validación tiene como objetivo servir de complemento a las pruebas de caja negra tradicionales. El entorno se puede usar para diferentes estaciones y para diferentes clientes. La validación del IM y del FEC para proyectos en España ha sido el primer paso para aplicar este método, que posteriormente se ha empleado también con otros clientes.

El objetivo principal de la estrategia de validación, que es encontrar fallos, se ha cumplido si bien no de la misma forma en los dos entornos. En el caso del IM, la forma de trabajo de lanzar todas las pruebas de una vez y luego analizar el resultado no se ha demostrado muy adecuada. La propia complejidad del IM y la cantidad de estados internos que se ha intentado verificar en la validación ha complicado mucho el análisis de los resultados de las pruebas cuando el entorno daba diferentes resultados que con las pruebas de caja negra manuales. La estrategia de validación ha dejado de lado

el modelar la interacción de diferentes elementos entre sí, y esto es una función bastante importante dentro del IM. Estas deficiencias han hecho que en el caso del IM el entorno de validación se haya rediseñado.

La estrategia de validación ha funcionado muy bien en el caso del entorno de validación del FEC. Se ha conseguido probar este subsistema de forma mucho más exhaustiva que con las pruebas manuales, que no obstante se continúan realizando por exigencias de la seguridad. Las entidades que gestiona el FEC (elementos de hardware) se modelan con gran exactitud mediante máquinas de estados y no hay apenas interacción de unos elementos con otros, es decir, que se puede probar cada elemento por separado sin necesidad de complicar el entorno de validación. El hecho de probar con los elementos de hardware reales hace que las pruebas vayan un poco más lentas al requerir la intervención del validador, pero por otra parte simplifica el entorno de validación.

Los requisitos generales del sistema en cuanto a rendimiento o disponibilidad no se cubren con esta estrategia de validación y requieren otro tipo de pruebas hecho a medida para verificarlos. En el caso de los requisitos de rendimiento, se prueba el sistema con una configuración donde la carga de trabajo es extrema y se le hacen pruebas manuales midiendo su tiempo de respuesta. La disponibilidad se determina siguiendo lo exigido por la norma CENELEC mediante un análisis FMEA (Fault Modes and Effects Analysis) que tiene como resultado un valor de MTBF (Mean Time Between Failures).

Los resultados obtenidos en cuanto a cobertura de requisitos han sido los establecidos por el estándar CENELEC: todos. Esto se ha conseguido combinando las pruebas automáticas y manuales. La trazabilidad de pruebas y de requisitos se ha establecido sin apenas soporte de herramientas. Al haber encontrado esta tarea manual muy tediosa, se propone su automatización en la medida de lo posible como línea de trabajo para el futuro: una automatización total de esta tarea no parece posible, ya que para poder asociar los requisitos a los elementos del diagrama de estados o del escenario hace falta conocer el dominio de aplicación en profundidad.

Los objetivos de cobertura de código se han cumplido parcialmente, debido a lo que ya se ha comentado antes. Los casos de prueba de validación están pensados para verificar el sistema en modo normal, y no cubren el tratamiento de excepciones, que supone una parte importante del código del IM y del FEC, como sistemas de seguridad que son. Por eso se ha preferido alcanzar este objetivo mediante pruebas unitarias.

La flexibilidad del entorno para adaptarse a diferentes clientes y estaciones se ha conseguido pero hasta la fecha no está soportada por ninguna herramienta y requiere todavía modificaciones manuales de los diferentes parámetros configurables del entorno (versiones de los ficheros con los casos de prueba, identificadores de los elementos, direcciones hardware, etc.). Esto trae el riesgo de una posible inconsistencia en el entorno cuando no todas las modificaciones se han hecho correctamente. En el caso del IM no se ha necesitado esta cualidad tanto como en el FEC, ya que se ha hecho la validación para un solo cliente. Aun así, el IM varía mucho más que el FEC pues es habitual que el cliente solicite para cada estación pequeñas variaciones a medida de la funcionalidad que son costosas de incorporar al entorno. Muchas veces se hacen sin apenas tiempo de ser reflejadas en la especificación de requisitos por falta de tiempo, lo cual hace muy difícil que se puedan incorporar al entorno automatizado y por eso se están probando hasta hora de forma manual. El soporte automatizado a la gestión de diferentes clientes es un aspecto mejorable en versiones futuras.

El tiempo invertido en realizar la primera versión de los dos entornos ha sido de tres personas/año a lo largo de un año y medio. Este tiempo se ha invertido en las tareas de análisis de la especificación de requisitos, modelado de las clases y subclases de prueba, análisis, diseño e implementación de los dos entornos de validación automatizados.

Aplicando esta estrategia de validación, es posible predecir cuanto se va a tardar en hacer las pruebas de un producto determinado, pues se sabe de entrada cuantos casos de prueba se han definido y lo que

se tardan en ejecutar. En el entorno de pruebas automatizado para el FEC es donde se han obtenido hasta la fecha los mejores resultados en cuanto a la relación del número de pruebas respecto al tiempo invertido en realizarlas. Se ahorra tiempo en la definición de las pruebas y en el registro de los resultados. Otro aspecto importante es que los validadores con poca experiencia reciben ayuda del entorno durante la ejecución de las pruebas y adquieren conocimiento del sistema. Los validadores más cualificados quedan descargados de estas pruebas más rutinarias y pueden dedicarse a otras más sofisticadas o diseñar pruebas específicas para cada producto que se salgan de lo que se puede probar con el entorno de validación.

5 Herramientas Software de Soporte del Método

5.1 Introducción

Como se ha dicho ya, una Línea de Productos software está compuesta por sistemas que tienen en común ciertas características en su arquitectura, sus requisitos, su dominio de aplicación, etc. El desarrollo de cada uno de los componentes de la Línea de Productos se facilita mediante el empleo de una base común para todos ellos. Para esto es necesario identificar en primer lugar los aspectos comunes a todos los sistemas e implementar mecanismos de generación de los productos específicos a partir de la base común. Las pruebas de estos sistemas de software también procuran seguir esa misma filosofía.

Si hubiera que resumir en una sola palabra lo que son las Líneas de Producto Software esta sería reutilización. Para conseguir esa reutilización en las pruebas se plantea, como ya se ha dicho, la alternativa de hacer las pruebas de la parte general una sola vez y darlas por pasadas en los demás productos, o bien, cuando esto no es posible, reutilizar todo lo que hay asociado a las pruebas: casos de prueba, planes de prueba, entornos de prueba, etc.

La automatización parcial o total de las pruebas es también una forma de conseguir la reutilización, y por tanto, de incrementar el rendimiento del proceso de pruebas de una línea de productos software. Los entornos automáticos sustituyen a las personas en la tarea de producir entradas en el sistema y analizar la corrección de las salidas en función de un resultado esperado. El mercado ofrece en la actualidad una gran variedad de herramientas que permiten obtener un cierto grado de automatización de las pruebas. Las principales ventajas de automatizar las pruebas son: proceso de pruebas definido, repetible y eficiente en cuanto a los costes. Pero estos beneficios no se obtienen sin haber realizado un esfuerzo previo en la definición, la implementación y el mantenimiento del entorno de pruebas automáticas.

Las pruebas automáticas y las Líneas de Producto tienen en común que consiguen mejorar la eficiencia del desarrollo de software. Se va a presentar ahora un conjunto de herramientas software que pretenden trabajar en ambas áreas. Las herramientas están diseñadas conforme a la estrategia de automatización de las pruebas de software en el desarrollo de Líneas de Producto que se ha expuesto en la parte metodológica de la Tesis Doctoral.

Este apartado de Herramientas de Soporte de la Tesis Doctoral comienza recordando los conceptos más importantes expuestos en la parte metodológica, para entender cómo se han aplicado posteriormente en un conjunto de herramientas software. Primeramente se explican las herramientas que se han utilizado en el caso práctico industrial, dejando de lado los aspectos confidenciales. Tras esta explicación, se exponen las características del entorno de herramientas para Prueba Automática de Líneas de Producto Software PLAT (Product Line Automated Testing) surgido en el ámbito académico en el entorno de la Tesis Doctoral, que busca añadir nuevas funcionalidades respecto del primer conjunto de herramientas y tener un carácter más genérico, aparte de servir para demostrar la aplicabilidad en la práctica de la aportación metodológica de la Tesis Doctoral. Se concluye haciendo mención de otras herramientas relacionadas surgidas en el ámbito académico.

5.2 Fundamentos

¿Tiene sentido hablar de unas pruebas específicas de Líneas de Producto Software? ¿En qué medida pueden volverse a utilizar las pruebas que se hacen en un producto en otro de la misma familia, de manera análoga a como se reutiliza la arquitectura? La respuesta a estas cuestiones depende del grado de modularidad de la Línea de Productos que se esté considerando (en qué medida la parte general está aislada de la parte dependiente de cada producto).

Otro factor a tener en cuenta es que las funcionalidades genéricas puedan identificarse claramente y se puedan probar independientemente de las funcionalidades generales. En algunos casos, el acoplamiento entre ambas partes hará imposible garantizar que las pruebas que se hayan hecho con éxito sobre un producto puedan obviarse para otro. Habría que plantearse entonces si el diseño del sistema es el correcto, pero entonces es posible que ya no se pueda cambiar. También puede suceder que el cliente, por los motivos que sea, obligue a realizar las mismas pruebas sobre cada producto de forma independiente. Por eso, el método de pruebas propuesto define una estrategia común a todos los componentes de la línea de productos que incluye la automatización de las pruebas para poderlas realizar de forma más eficiente.

Como se ha dicho ya, un concepto muy importante en el método de pruebas propuesto en la Tesis Doctoral es mantener la consistencia (trazabilidad) en la organización de los requisitos, las variaciones en la arquitectura y los conjuntos de casos de prueba. Esta consistencia se garantiza asociando todas estas entidades con los requisitos del sistema. Los requisitos son el punto de partida de las pruebas. Un caso de prueba genérico se asocia con un requisito genérico de la línea de productos, y a un caso de prueba específico de un producto le corresponde un requisito particular de un producto. De esa forma se puede determinar cuántos requisitos se prueban con un conjunto de casos de prueba dado.

Pero se ha dicho que el método de pruebas además de trazabilidad, tiene como objetivo lograr una automatización de las pruebas. Para eso, se necesitan modelos formales o semiformales de comportamiento del sistema, que tengan las siguientes características:

- Reflejan fielmente la implementación que se va a probar.
- No contienen detalles irrelevantes, pero tiene el suficiente grado de detalle para garantizar que no se queden sin probar requisitos del sistema.
- Representan todos los eventos posibles, tanto internos como externos.
- Ofrecen mecanismos para comprobar cómo están transcurriendo las pruebas.

En este caso, como se explica en la parte metodológica de la Tesis Doctoral, se ha elegido utilizar los diagramas de estados jerárquicos de Harel [Har88], o statecharts, que forman parte del lenguaje UML, y que amplían la capacidad de los diagramas de estados tradicionales. Es una técnica que no sólo puede usarse en el diseño y la implementación, sino también en las pruebas.

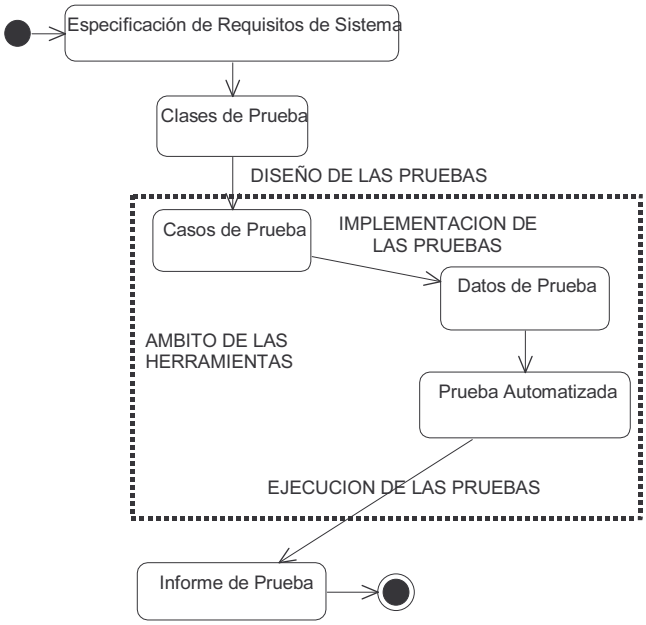


Figura 5.1 Actividades a partir del diseño de las pruebas hasta su realización

La Figura 5.1 muestra el flujo de actividades que se definen para la automatización de las pruebas, marcando el ámbito de aplicación de las herramientas de soporte en todo el proceso. Los requisitos del sistema son la base del proceso. Se clasifican en clases de prueba, que se modelan en forma de statechart de forma manual. Se asocia, también manualmente, y este es un paso del método que requiere un gran esfuerzo, a cada transición del modelo de estados requisitos, como muestra la Figura 5.2, para poder medir cuantos requisitos se prueban explorando todo el diagrama completo. A partir de este modelo se obtienen los casos de prueba, ya de forma automatizada si se cuenta con herramientas de soporte que "recorran" el modelo de máquina de estados de acuerdo a una determinada estrategia y elaboren una lista de casos de prueba. Estos casos de prueba se implementan, también con ayuda de herramientas de soporte, mediante los datos de prueba y se pueden ejecutar en el entorno de pruebas automáticas.

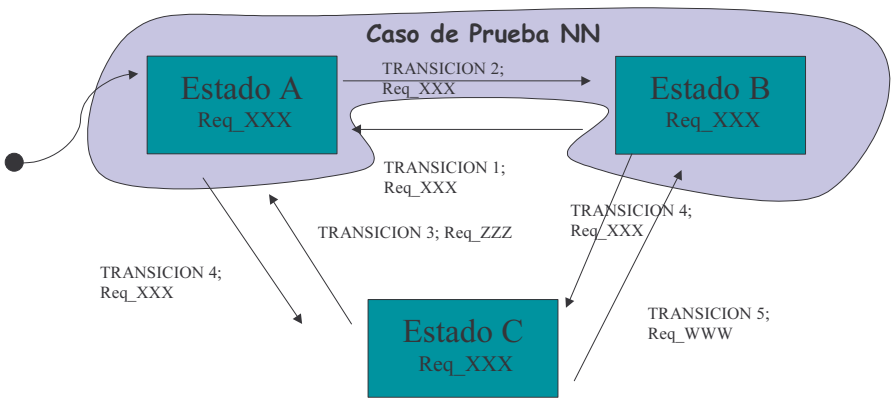


Figura 5.2 Diagrama de estados con información de requisitos

En la Figura 5.3 se muestra el flujo de actividades de prueba, añadiendo la confección de los casos de prueba genéricos y específicos para la Línea de Producto Software, y mostrando la trazabilidad que

acompaña todo el proceso. En el ámbito de aplicación de las herramientas de soporte están incluidas la generación de casos de prueba a partir de los modelos de prueba (máquinas de estados). Conociendo a partir de qué modelo de estados se han generado los casos de prueba, se pueden organizar en clases, y sabiendo si el diagrama de estados es genérico o específico, tenemos esa misma división en los casos de prueba. Tras obtener los casos de prueba, las herramientas de soporte sirven también de ayuda para obtener los datos de prueba ejecutables en el entorno de pruebas automáticas.

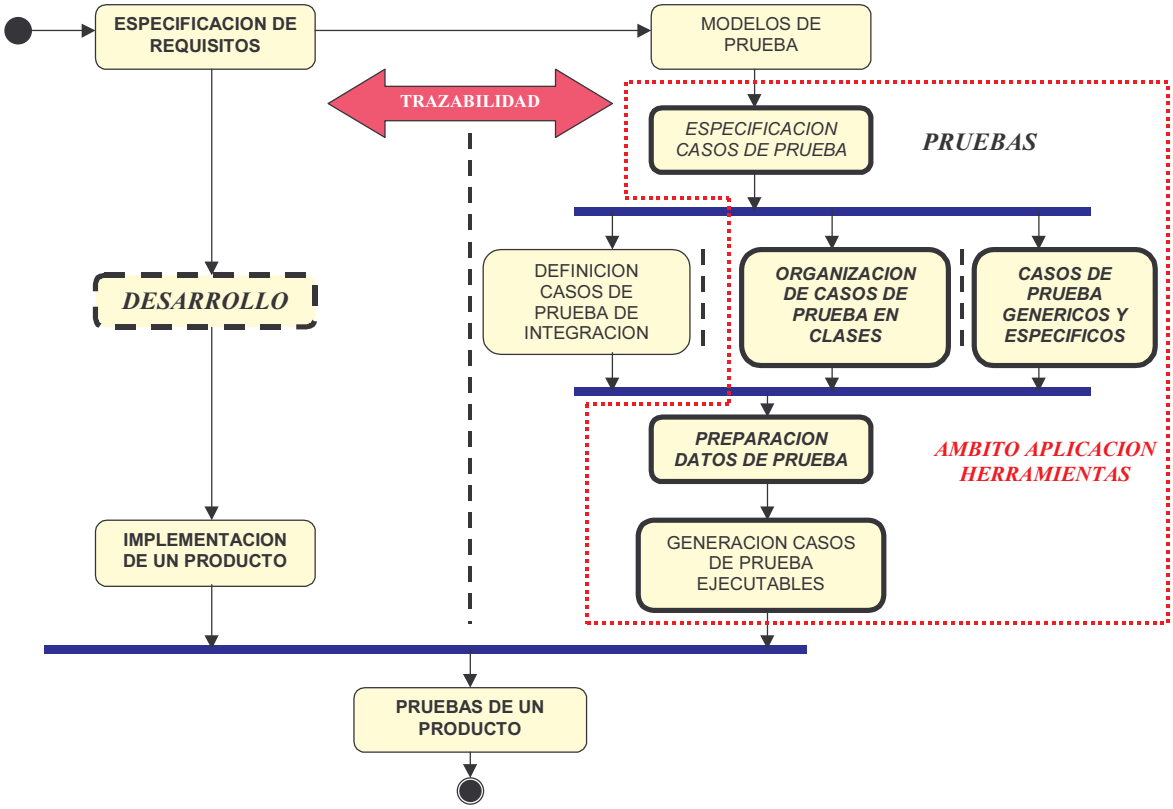


Figura 5.3 Area de aplicación de las herramientas dentro del Flujo de Actividades

5.3 Herramientas aplicadas en el caso de validación industrial

Como se ha relatado ya, en el caso industrial de aplicación del método de pruebas, se utilizaron una serie de herramientas software de carácter propietario dentro de la compañía Alcatel, en cooperación con la consultora internacional especializada en procesos de prueba SQS. En este apartado se explican, dentro de las limitaciones que impone la confidencialidad, las características de dichas herramientas [EMM02][MeET02].

Las herramientas que se explican son: el generador de casos de prueba “Test Composer”, el ejecutor de Pruebas “Test Tracker”, el emisor de veredictos de prueba “Test Comparer” y el medidor de cobertura de Pruebas “Test Cover”. En la Figura 5.4 se muestra en qué fase concreta del proceso de automatización se ha usado cada herramienta:

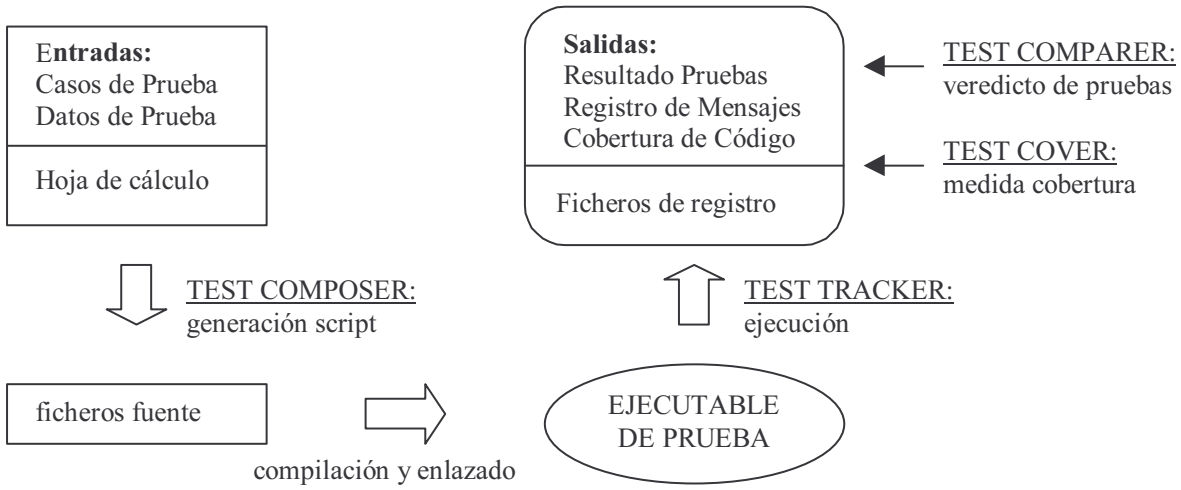


Figura 5.4 Herramientas aplicadas en el ejemplo industrial

5.3.1 Generador de Casos de prueba "Test Composer"

Esta herramienta genera los casos de prueba para el diagrama de estados a partir de la información contenida en una hoja de cálculo comercial (Excel) sobre los cambios de estado. Cada hoja de cálculo especifica una clase de prueba, es decir, una máquina de estados en forma de una tabla de estados y eventos, que se crea manualmente. Como se ha dicho ya, este concepto se ha aplicado con algunas diferencias sobre dos subsistemas diferentes, el FEC y el IM, que tienen diferente grado de complejidad en cuanto al número de estados y de transiciones posibles. La respuesta de la máquina de estados al evento exterior puede ser un cambio de estado, o un cambio de estado y el envío de un mensaje al exterior. Las máquinas de estados especificadas son máquinas de estados tradicionales, es decir no tienen jerarquías de estados. Por motivos de la certificación de seguridad de los subsistemas bajo prueba, asociada a las tablas de estados se incluye información de requisitos con el objetivo de documentar la trazabilidad, pero esa información se introduce en la hoja de cálculo de forma manual, lo que requiere un esfuerzo para mantener la información en un estado consistente a lo largo de todo el proceso. En el caso del entorno para el IM la información de trazabilidad de requisitos se introduce en una hoja de cálculo separada, y en el caso del FEC, con la experiencia adquirida en el entorno anterior, se usa una única hoja de cálculo para toda la información de la máquina de estados.

El caso de prueba se define de la forma:

$$CP = \{Identificador, E_{inicial}, Evento, E_{final}\}$$

Cada caso de prueba tiene un identificador que es usado por las diferentes herramientas del entorno para gestionarlo, y que se usa también para monitorizar el avance de las pruebas. Los estados inicial y final corresponden a estados de la clase de prueba correspondiente. El evento es en ambos entornos un mensaje, en el caso del entorno del IM puede ser un mensaje del FEC o del OM, y en el caso del entorno del FEC es siempre un mensaje del IM.

En la hoja de cálculo aparte de la información de la máquina de estados hay que incluir la información para los datos de prueba. Es decir, que hay que indicar valores concretos, también de forma manual, para los parámetros de configuración tanto del FEC como del IM. Recuérdese que se ha dicho que la variabilidad de estos subsistemas se implementa mediante ficheros de configuración específicos para cada instalación real y específicos para cada cliente (administración ferroviaria, en este caso). Una

mejora del entorno sería que los datos de prueba se reconfiguraran de forma automática leyendo los ficheros de configuración originales de cada instalación, pero no se ha abordado dentro de la Tesis Doctoral por estar muy ligada al caso concreto de aplicación.

En ambos entornos la estrategia de generación de pruebas es la de recorrer transiciones simples; es decir, no se contemplan transiciones que pasen por varios estados. Los casos en los que esta estrategia es insuficiente, se cubrieron con escenarios ejecutados de forma manual. De ahí surgió la idea de desarrollar una nueva herramienta en el marco de la Tesis Doctoral para mejorar esta fase del proceso, que se explica en el apartado 5.4.4.

En ambos entornos de validación, una vez que la hoja de cálculo con la tabla de estados se ha completado, se arranca la herramienta "Test Composer", se selecciona la clase de prueba a la que corresponde la hoja de cálculo y el "Test Composer" transforma los casos de prueba en código, que se integra en el entorno de prueba automatizada. Aquí es claro que se está asumiendo la existencia de puntos de acceso a la implementación (IAP) en los subsistemas bajo prueba que permiten la interacción del código ejecutable generado.

5.3.2 Ejecutor de Pruebas "Test Tracker"

Esta herramienta sólo se utiliza en el entorno de validación del FEC y sirve para realizar las pruebas de forma interactiva. El "Test Tracker" se ejecuta en una plataforma hardware diferente (un PC con Windows concretamente) que la de la implementación bajo prueba, el FEC, que corre en una plataforma hardware empotrada específica, porque esto descarga al sistema bajo prueba, el FEC, del tiempo de proceso necesario para dialogar con el operador, que para un sistema de tiempo real como el FEC es difícilmente asumible y proporciona además una interfaz de usuario gráfica de la que el FEC no dispone.

En el caso del entorno de validación del IM, la ejecución de las pruebas se realiza en modo "batch" o secuencial en la misma plataforma hardware que el sistema real, y no existe un "Test Tracker", teniendo esto la ventaja de que se realizan de una vez un número muy elevado de casos de prueba, pero también con la desventaja de que durante la fase de depuración del entorno automático, ha sido preciso analizar manualmente ficheros de registro de gran tamaño, para asegurar que el entorno no estaba falseando los resultados.

La forma de operar el "Test Tracker" se describe a continuación.

El operador selecciona en un menú primero la clase de prueba y luego los casos de prueba que va a ejecutar, conforme al plan de pruebas de validación, por ejemplo todos los casos de prueba de las agujas para RENFE.

- El "Test Tracker" ejecuta entonces en secuencia todos los casos seleccionados:

    - Primero se asegura que el estado del elemento hardware es el que se ha definido para el caso de prueba. En el caso del ejemplo de la aguja de tipo RENFE, el operador comprobaría el estado del relé de posición de aguja en el grupo de relés que se conecta al motor de la aguja. Supongamos que se ha seleccionado el caso de prueba de mover aguja partiendo de posición a derechas, que es la posición en la que mirando desde la punta de la aguja, se lleva al tren a cambiar de dirección a la derecha.

    - Si el estado inicial no es el correcto, el "Test Tracker" pide al operador su intervención para cambiarlo. El "Test Tracker "se puede configurar para que genere un mando sobre el FEC para cambiar de estado al elemento, o bien para que el operador cambie el estado del elemento

a mano y dé su visto bueno para comenzar a ejecutar el caso de prueba. En el caso del ejemplo, si la aguja en vez de estar a derechas estaba a izquierdas, el operador le dice al "Test Tracker" que el estado inicial no es correcto y el "Test Tracker" envía al FEC un mando para cambiar la aguja a posición derecha.

- En este momento tiene lugar la ejecución del caso de prueba, que puede también requerir la confirmación y la intervención manual del operador. En el caso del ejemplo, es totalmente automático, el "Test Tracker" envía al FEC el mando de mover aguja, y si el FEC mueve la aguja a derechas, se lo notifica al "Test Tracker", que da el caso de prueba por pasado. Si hay algún problema de hardware o de software por el que la aguja no se haya movido, el "Test Tracker" lo notifica al operador y le pide que confirme el fallo.

- El "Test Tracker" registra el resultado de la prueba y pasa a un nuevo caso de prueba. Los resultados de todos los casos de prueba quedan almacenados en un fichero de registro, disponibles para la medida de la cobertura y el seguimiento del proceso de pruebas.

- Cuando la ejecución de los casos de Prueba ha terminado, el operador puede seleccionar más casos de prueba o interrumpir la ejecución, si lo desea.

5.3.3 Emisor de Veredictos de Prueba "Test Comparer"

Esta herramienta analiza los ficheros de registro creados durante la generación de las pruebas y los analiza. El entorno del IM tiene un "Test Comparer" diferente al del FEC, pues los contenidos de los ficheros de registro son diferentes (ver ejemplos en Figura 4.28 y Figura 4.30). En el entorno del FEC, el "Test Tracker" ya ha emitido el veredicto de prueba y la tarea del "Test Comparer" es elaborar a partir del fichero de registro creado por el "Test Tracker" una lista con todos los casos de prueba ejecutados, diciendo cuáles han pasado y cuáles han fallado. En el caso del IM no existe "Test Tracker" sino que la ejecución es secuencial en modo no interactivo, y se guarda en el fichero de registro el resultado real del caso de prueba y el resultado esperado del caso de prueba para que el "Test Comparer" los analice. El "Test Comparer" analiza en el fichero de registro si el resultado real se corresponde con el esperado, y si es así emite un veredicto de caso de prueba pasado, y si no, de caso de prueba fallido (Ver Figura 4.29).

En ambos entornos, el operador selecciona en un menú el fichero de registro de pruebas y el "Test Comparer" crea un nuevo fichero de resultados con la lista de todos los casos de prueba ejecutados, y una estadística final de casos de prueba pasados y casos de prueba fallidos.

5.3.4 Evaluador de Cobertura de Pruebas "Test Cover"

Para saber qué porcentaje de los requisitos se ha cubierto con las pruebas realizadas, se usa la herramienta "Test Cover". Esta herramienta se sirve de la información de trazabilidad de requisitos existentes en las hojas de cálculo donde se especifican las máquinas de estados en forma de tabla.

Como ya se ha dicho, se asocia a cada caso de prueba uno o más requisitos, tras analizar el contenido de la especificación, lo que requiere esfuerzo, y no es automatizable. Este esfuerzo crece mucho si los requisitos son muy numerosos y tienen un nivel de detalle muy alto, por ello en ese caso una opción a considerar es la de clasificar los requisitos en "niveles de especificación", por ejemplo unos que sean esenciales y otros de detalle, y asignar a las transiciones y estados del modelo de la clase de prueba sólo los requisitos esenciales. En el caso de validación industrial el esfuerzo de trazabilidad está justificado por las exigencias de certificación de seguridad del dominio de aplicación.

El "Test Cover" necesita como entrada las hojas de cálculo que contienen los casos de prueba. El operador selecciona la hoja de cálculo correspondiente a la clase de prueba que desea verificar y el "Test Cover" genera un fichero donde aparecen todos los requisitos asociados a los casos de prueba seleccionados. Adicionalmente, el operador dispone de una serie de comandos de consulta para obtener los valores del número de casos de prueba y el número de requisitos cubiertos.

La herramienta "Test Cover" no se utilizó de forma tan intensiva como las demás, pues no se consiguió por motivos de tiempo una versión que pudiera integrar de forma automática los resultados de todas las clases de prueba de una vez, y esa labor debió de ser realizada manualmente integrando la información de todos los ficheros de registro de la ejecución de los casos de prueba para ambos entornos automáticos.

5.4 Entorno de herramientas PLAT *(Product Line Automated Testing*)

5.4.1 Visión Global

El método de pruebas que se propone en la Tesis Doctoral hace hincapié en la trazabilidad de requisitos, casos de prueba y variaciones en la línea de productos. Esa trazabilidad, que equivale a consistencia, se consigue por medio de diversas técnicas, que se han explicado en la parte de metodología de la Tesis Doctoral. Este apartado explica cómo las herramientas que se han ideado para dar soporte a este método están relacionadas entre sí formando un entorno de prueba software denominado PLAT (*Product Line Automated Test Toolchain*) y que está compuesto por:

- Herramienta de medición de cobertura de Requisitos*, Product Line Oriented Requirements Coverage Manager (PLOR)*. Esta herramienta determina qué porcentaje de los requisitos queda cubierto por un determinado conjunto de casos de prueba.

- Herramienta de generación de diagramas de estados, *Product Line Oriented Statechart Generator (PLOS)*. Esta herramienta permite elaborar diagramas de estados jerárquicos (statecharts), con partes genéricas que luego dan lugar a diferentes variantes de un mismo diagrama. Esta herramienta ya existe como proyecto de investigación en el DIT de la UPM, [Ce01] y se ha visto que podría integrarse dentro del entorno PLAT.

- Herramienta de construcción de casos de prueba, *Product Line Oriented Test Builder (PLOT).* Esta herramienta construye a partir de un statechart casos de prueba. Los statecharts genéricos son abstracciones que no son directamente susceptibles de ser probadas y por esta razón se ha decidido que la herramienta PLOT utilice como entrada los statecharts que están implementados en los distintos productos finales. La herramienta PLOT produce un conjunto de caso de prueba a partir de estos diagramas que están definidos de acuerdo con la metodología propuesta, por su estado inicial, su estado final, un evento que produce la transición del estado inicial al estado final y un conjunto de requisitos asociados con la transición.

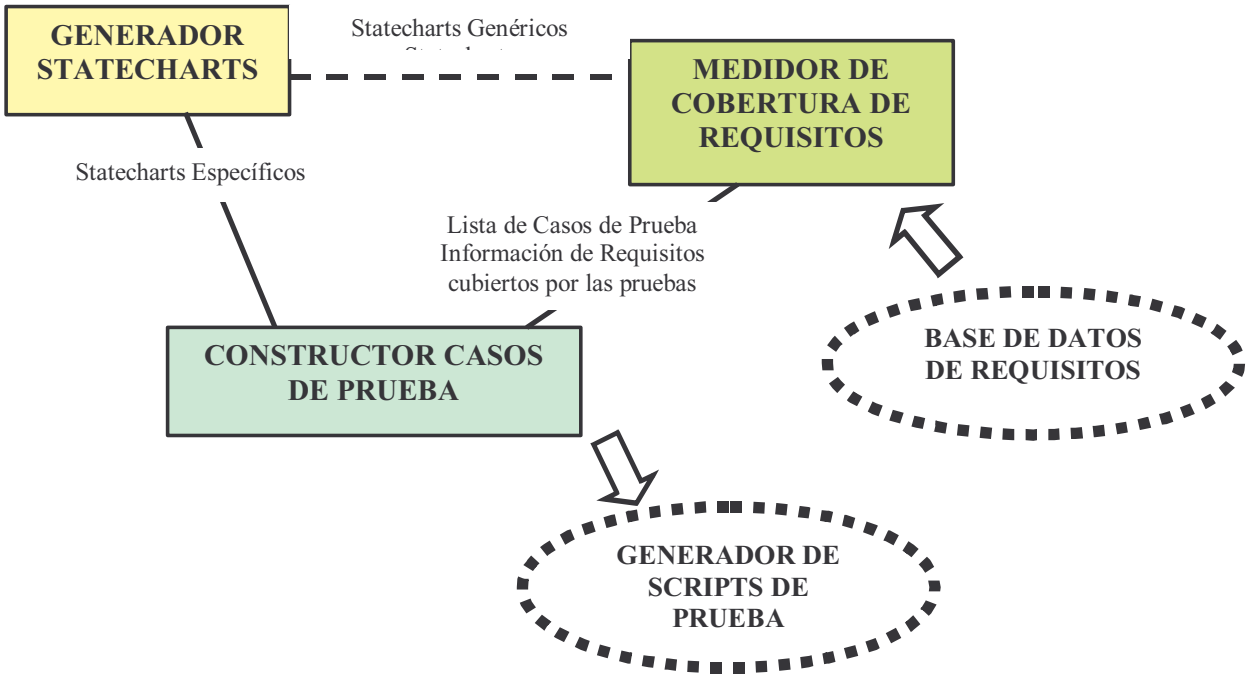


Figura 5.5 Esquema general del Entorno PLAT

El esquema representa las relaciones entre las diferentes herramientas software del entorno. El generador de statecharts sirve para modelar la parte genérica de la línea de productos, que abarca todos aquellos requisitos que estén presentes en todos los componentes de la familia de productos. La consistencia entre los statecharts genéricos y los requisitos genéricos está representada por una línea discontinua en el esquema que une el generador de statecharts con el gestor de cobertura de requisitos. El Generador produce como salida los diagramas de estados que se implementan en cada componente de la familia de productos. Estos diagramas son los que el Constructor de Casos de Prueba necesita como dato de entrada. La salida del Constructor de Casos de Prueba es un conjunto de Casos de Prueba que cubren todo el diagrama de estados y que incluyen además información de los requisitos que se están comprobando con cada uno de dichos casos de prueba. Esto está representado mediante una línea continua en la figura. El Medidor de Cobertura de requisitos utiliza esta información para compararla con todo el conjunto de requisitos de la línea de productos. El resultado de esta comparación es una medida de cobertura de requisitos de los casos de prueba, en forma de porcentaje de requisitos comprobados sobre el total.

En el esquema aparecen también representadas otras dos herramientas que no forman parte del entorno, pero que se asume que existen. La primera es un generador de scripts de prueba. Utiliza la salida del Test Case Builder en forma de tabla y, dependiendo de la implementación bajo prueba, traduce esos casos de prueba en las llamadas a función correspondientes. La segunda es una base de datos que contiene los requisitos y a la que se hacen consultas para determinar los requisitos del sistema. Lo único que hace falta es que exista un identificador individual para cada requisito y un campo que diga si el requisito es genérico o específico.

El entorno de herramientas software PLAT utiliza como base otros productos software comerciales. El generador de Statecharts usa los servicios de la herramienta de modelado con el lenguaje UML Rational Rose. El Generador de Casos de prueba usa también Rational Rose, y la hoja de cálculo Microsoft Excel. El gestor de cobertura de requisitos usa la hoja de cálculo Microsoft Excel.

El generador de Statecharts se ha desarrollado por parte del DIT de la UPM y se encuentra descrito en [Cer01]. El generador de casos de prueba se ha desarrollado en un proyecto de fin de carrera y es la herramienta que se va a exponer con mayor detalle. El medidor de Cobertura de Requisitos se ha desarrollado por el autor de la Tesis Doctoral, y es un pequeño añadido sobre Excel (inicialmente se probó a usar Access, pero Excel dio mejores resultados) para generar los datos de cobertura a partir de una hoja de cálculo con los casos de prueba ya realizados, aunque es posible que una versión mejorada sea materia de otro proyecto de fin de carrera. El objetivo que se ha fijado a la hora de hacer estas herramientas no es el de desarrollar productos comerciales, ya que para eso hubieran sido necesarios muchos más recursos que los empleados. Con estas herramientas prototipo se ha buscado explorar las posibilidades existentes para ver cómo pueden solucionarse con los medios actuales que ofrece la tecnología los problemas, que se presentan en las pruebas de Líneas de Producto y, sobre todo, se ha pretendido validar la aportación metodológica de la Tesis Doctoral demostrando que se puede aplicar en la práctica.

5.4.2 Herramienta de medición de cobertura de Requisitos, Product Line Oriented Requirements Coverage Manager (PLOR)

Función	Mide la cobertura proporcionada por parte de un conjunto de casos de prueba respecto del conjunto completo de los requisitos
Entradas	• Lista de Casos de Prueba con los requisitos cubiertos • Lista de Requisitos
Salidas	• Medida de Cobertura de requisitos
Interfaces	• Lista de Requisitos, con información de cuáles de los requisitos son genéricos, y cuáles específicos de un producto. • Constructor de Casos de Prueba PLOT
Herramienta Base	Microsoft Access, Microsoft Excel

5.4.3 Herramienta de generación de diagramas de estados, Product Line Oriented Statechart Generator (PLOS)

Función	Relaciona los diagramas de estados genéricos que son válidos para toda la línea de productos con los diagramas de estados propios de cada producto individual: • Los estados genéricos se convierten en estados particulares de un producto • Las transiciones genéricas se transforman en transiciones específicas a los estados particulares.
Entradas	• Statecharts Genéricos

Salidas	• Statecharts derivados a partir de los Statecharts Genéricos
Interfaces	• Constructor de Casos de Prueba PLOT
Herramienta Base	Rational Rose

5.4.4 Herramienta de construcción de casos de prueba, Product Line Oriented Test Builder (PLOT)

Función	Analiza un statechart construyendo a partir de él un conjunto completo de casos de prueba. Para eso utiliza dos estrategias. La primera de ellas construye un conjunto de casos que cubre por completo todos los estados y transiciones del diagrama elaborando un *árbol de transiciones* según una estrategia de pruebas N+. La segunda estrategia asegura que no hay caminos ilegales en la implementación del diagrama de estados comprobando para cada estado todos los eventos posibles.
Entradas	• Statechart conteniendo información sobre los requisitos que modela. Esto se hace con los campos de información adicional que ofrece la herramienta Rational Rose.
Salidas	• Prueba de Recorrido de la máquina de estados • Arbol de Transiciones • Prueba de Caminos Ilegales
Interfaces	• Rational Rose Statecharts (pueden realizarse directamente con Rose o ser producto del Generador de Statecharts PLOS) • Generador de Scripts de Prueba • Medidor de Cobertura de Requisitos PLOR
Herramienta Base	Rational Rose

El proceso de trabajo con la herramienta PLOT está representado en la Figura 5.6. Se parte de una Línea de Productos software, que tiene unos requisitos, algunos comunes a todos los productos y otros específicos de algunos de ellos. Se modelan los requisitos con diagramas de estados jerárquicos (statecharts) que se convierten mediante la herramienta PLOT en tablas y a partir de las cuales la herramienta PLOT genera por este orden, el árbol de estados, la prueba de recorrido de la máquina de estados y los caminos ilegales. Los casos de prueba se ejecutan y se obtienen unos resultados (caso de prueba pasa, caso de prueba no pasa) junto con una medida de cobertura de las pruebas (porcentaje de requisitos probados respecto del total).

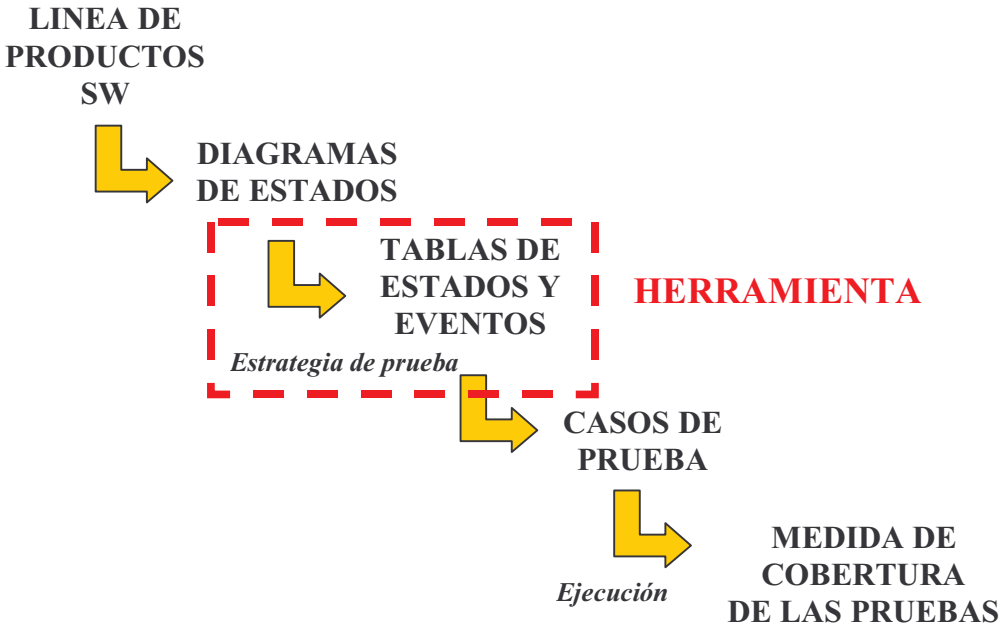


Figura 5.6 Proceso de prueba

La implementación de la herramienta PLOT se ha realizado en Visual Basic y los diagramas de estados jerárquicos se dibujan con la herramienta comercial Rational Rose. Por último, los casos de prueba se van a generar en un fichero Excel, siguiendo una estrategia N+.

5.4.4.1 Funcionamiento de la herramienta PLOT

El funcionamiento de la herramienta PLOT tiene tres actividades claramente diferenciadas: (1) extraer la información del diagrama, (2) manipular los datos obtenidos y (3) presentar los resultados de manera legible para el usuario.

- Extraer la información. Esta parte no ofrece grandes problemas, ya que el fichero Rose que contiene el diagrama es fácilmente exportable a Excel y presenta un formato en el que los datos vienen precedidos por un identificador, por lo que recorriendo las celdas del fichero puede extraerse toda las referencias necesarias a los estados y transiciones que forman el diagrama. Por supuesto, no basta sólo con extraerla, sino que es preciso también guardarla para cuando sea necesaria, por lo que se han utilizado unas estructuras en forma de tablas en las que se van almacenando todos los datos.

- Manipular los datos. Esta ha sido quizá la parte más compleja de realizar. En esta fase se desarrollan cada uno de los pasos de la estrategia N+. El programa, partiendo del diagrama de estados elabora dos tablas, una de transiciones y otra que incluye los requisitos asociados. Una vez hecho esto, se pasa a una representación en forma de árbol, lo que permite probar todos los posibles caminos desde el estado inicial hasta un estado final, entendido éste como un estado por el que ya se ha pasado. Se ha tenido que diseñar una estructura de datos para poder realizar esta parte, ya que la representación en tablas no era la más adecuada. Una vez hecho esto se obtiene de

forma sencilla una tabla en la que se presentan todos los caminos posibles y la forma de recorrerlos y además permite asegurar que no es posible recorrer un camino de una forma que no se haya definido.

- Presentar los datos. El programa toma todos los datos que ha obtenido y que se encuentran en un almacén y presenta los resultados en una hoja de Excel para que sean más fácilmente comprensibles.

5.4.4.2 Descripción del programa

La herramienta PLOT es *una herramienta que genera casos de prueba ejecutables a partir de un diagrama de estados*. Es decir, un programa cuya entrada es la especificación de los requisitos del sistema software expresados en forma de diagrama de estados jerárquicos y cuya salida es un conjunto de casos de prueba que pueden realizarse de forma automática.

La herramienta toma como punto de partida un diagrama de estados generado en Rose; que es leído y analizado por una aplicación en Visual Basic generando un fichero Excel en el que se muestran:

- tablas de transiciones,
- trazabilidad de requisitos con casos de prueba,
- árbol de estados,
- prueba de recorrido de la máquina de estados y
- prueba de caminos ilegales.

En líneas muy generales puede describirse el programa con el diagrama que aparece en la Figura 5.7.

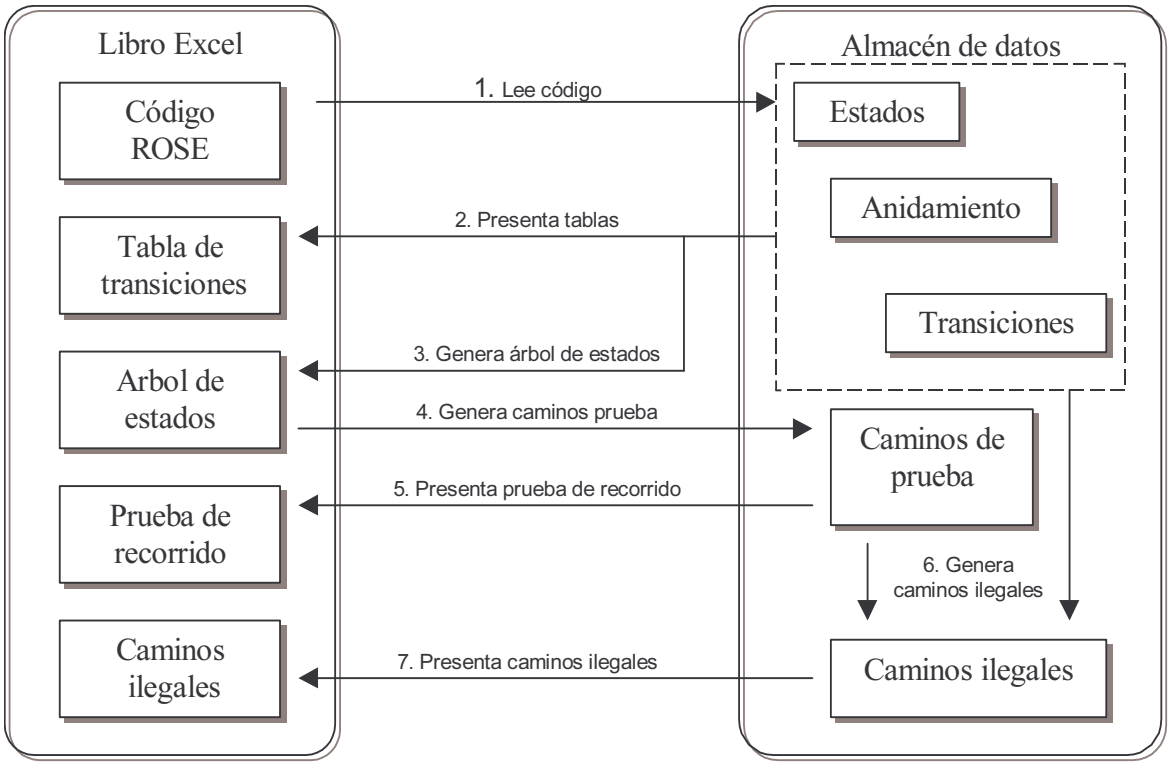


Figura 5.7 Partes funcionales del programa

Partiendo de una hoja Excel que contiene el código Rose del diagrama se generan (1) las tablas de estados y transiciones que se almacenan en variables. De estas tablas se obtienen dos presentaciones en hojas Excel diferentes: (2) en forma de tablas y (3) en forma de árbol. Del árbol representado en Excel se obtiene (4) una tabla con la prueba de recorrido de la máquina de estados, que se muestra (5) en una nueva hoja. Por último, combinando (6) el contenido de las variables 'tablas de estados', 'tabla de transiciones' y 'caminos de prueba' se obtiene el listado de los caminos ilegales que también se muestra (7) en hoja aparte.

5.4.4.3 Casos de estudio

En esta sección se van a mostrar dos diagramas de estados, uno simple y otro con estados jerárquicos, analizados con la herramienta PLOT. En los dos casos se presentara en primer lugar el diagrama y a continuación los datos generados con la herramienta.

Diagrama de Estados Simple

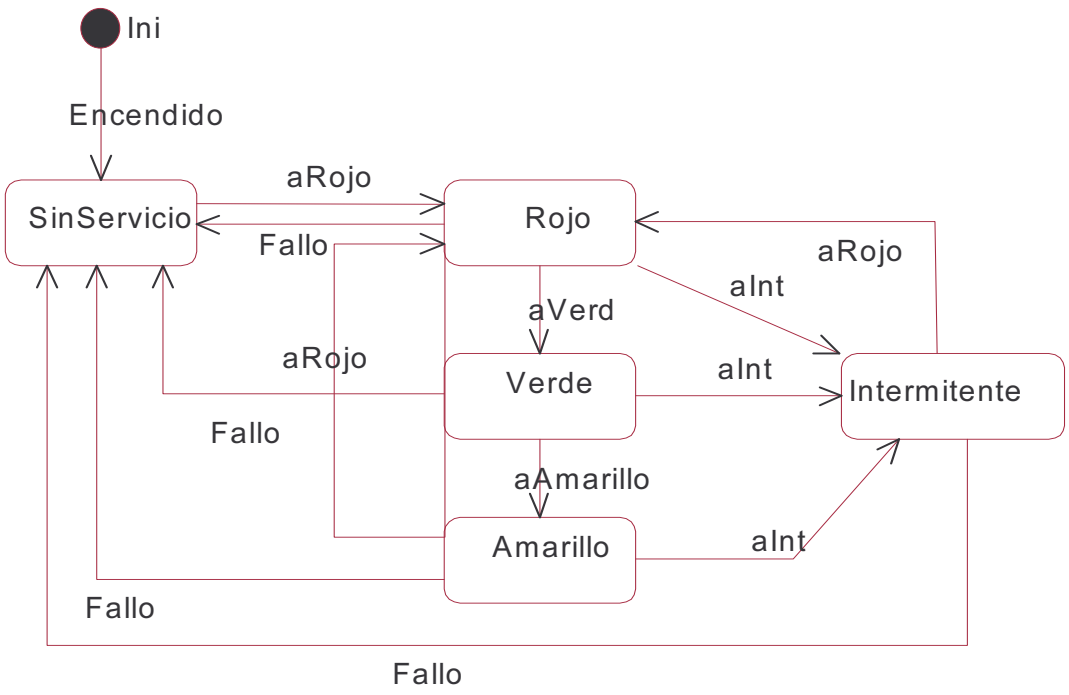


Figura 5.8 Diagrama de estados

El diagrama representa el funcionamiento de un semáforo que, al encenderse, no se pone en servicio directamente sino que se lleva a cabo una revisión del sistema. Sólo si el resultado es el esperado se pone en servicio comenzando por iluminarse el rojo, sigue el juego de luces típico rojo – verde - amarillo, desde cualquiera de ellos se puede pasar a la luz roja intermitente, desde la que sólo puede pasarse a rojo, y en caso de fallo desde cualquier estado se queda fuera de servicio, hasta que se efectúa de nuevo la revisión.

A continuación se pueden ver las tablas de requisitos correspondientes:

estado	requisitos		
Ini			
SinServicio			
Rojo	RdLuzRoja	RdCiclo	RdUso
Verde	RdLuzVerde		
Amarillo	RdLuzAmarilla		
Intermitente			

transición	requisitos
Encendido	
ResetOK	
aInt	RdInt
aAmarillo	CambioLuzA
aRojo	CambioLuzR
aVerde	CambioLuzV
Fallo	rError

Tabla 5.1 Requisitos de los estados y de las transiciones

		Encendido	aAmarillo	aInt	Fallo	ResetOK	aRojo	aVerde
Ini	Inicial	SinServicio						
Verde	Normal		Amarillo	Intermitente	SinServicio			
SinServicio	Normal					Rojo		
Amarillo	Normal			Intermitente	SinServicio		Rojo	
Intermitente	Normal				SinServicio		Rojo	
Rojo	Normal			Intermitente	SinServicio			Verde

Tabla 5.2 Tabla de transiciones

		Encendido	aAmarillo	aInt	Fallo	ResetOK	aRojo	aVerde
Ini								
Verde	; RdUso; RdCiclo; RdLuzVerde		CambioLuzA	RdInt	rError			
SinServicio								
Amarillo	; RdUso; RdCiclo; RdLuzAmarilla			RdInt	rError		CambioLuzR	
Intermitente	; RdUso;				rError			
Rojo	; RdUso; RdCiclo; RdLuzRoja			RdInt	rError			CambioLuzV

Tabla 5.3 Tabla de requisitos

Se presentan a continuación el árbol de estados, la prueba de recorrido de la máquina de estados, y los caminos ilegales.

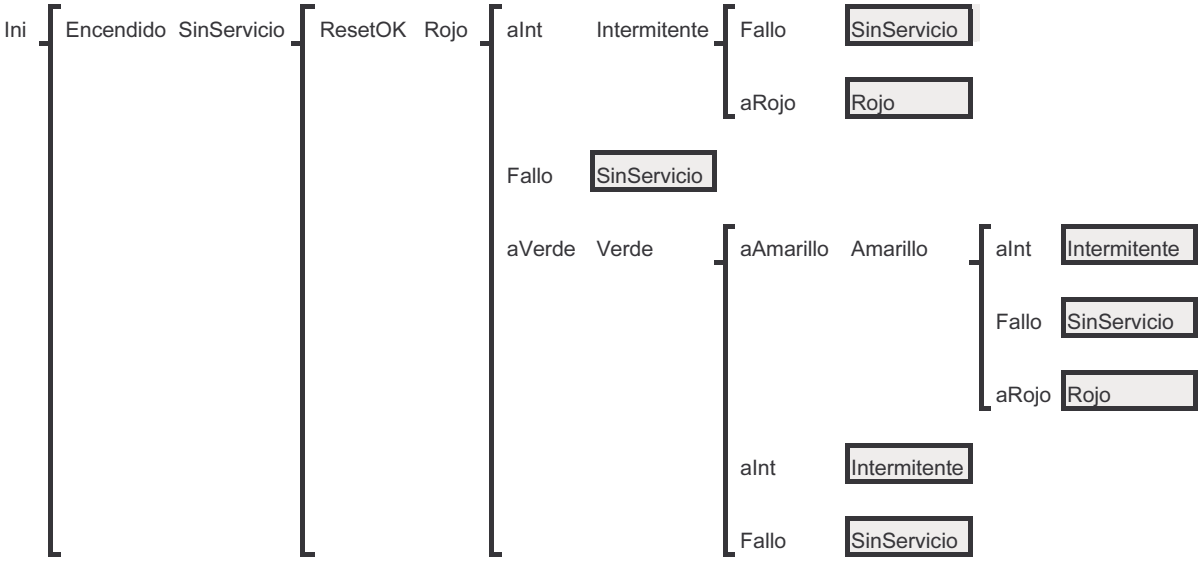


Tabla 5.4 Arbol de estados

ID	Evento	Estado esperado
1. 1	Encendido	SinServicio
1. 2	ResetOK	Rojo
1. 3	aInt	Intermitente
1. 4	Fallo	SinServicio
2. 1	Encendido	SinServicio
2. 2	ResetOK	Rojo
2. 3	aInt	Intermitente
2. 4	aRojo	Rojo
3. 1	Encendido	SinServicio
3. 2	ResetOK	Rojo
3. 3	Fallo	SinServicio
4. 1	Encendido	SinServicio
4. 2	ResetOK	Rojo
4. 3	aVerde	Verde
4. 4	aAmarillo	Amarillo
4. 5	aInt	Intermitente

ID	Evento	Estado esperado
5. 1	Encendido	SinServicio
5. 2	ResetOK	Rojo
5. 3	aVerde	Verde
5. 4	aAmarillo	Amarillo
5. 5	Fallo	SinServicio
6. 1	Encendido	SinServicio
6. 2	ResetOK	Rojo
6. 3	aVerde	Verde
6. 4	aAmarillo	Amarillo
6. 5	aRojo	Rojo
7. 1	Encendido	SinServicio
7. 2	ResetOK	Rojo
7. 3	aVerde	Verde
7. 4	aInt	Intermitente
8. 1	Encendido	SinServicio
8. 2	ResetOK	Rojo
8. 3	aVerde	Verde
8. 4	Fallo	SinServicio

Tabla 5.5 Prueba de recorrido de la máquina de estados

ID	Caso de prueba			Resultado esperado
	Inicialización	Estado	Evento	
CI.1.0	Estado por defecto	Ini	aAmarillo	Excepción Evento Ilegal
CI.1.1	Estado por defecto	Ini	aInt	Excepción Evento Ilegal
CI.1.2	Estado por defecto	Ini	Fallo	Excepción Evento Ilegal
CI.1.3	Estado por defecto	Ini	ResetOK	Excepción Evento Ilegal
CI.1.4	Estado por defecto	Ini	aRojo	Excepción Evento Ilegal
CI.1.5	Estado por defecto	Ini	aVerde	Excepción Evento Ilegal
CI.2.0	4. 1, 4. 2, 4. 3	Verde	Encendido	Excepción Evento Ilegal
CI.2.1	4. 1, 4. 2, 4. 3	Verde	ResetOK	Excepción Evento Ilegal
CI.2.2	4. 1, 4. 2, 4. 3	Verde	aRojo	Excepción Evento Ilegal
CI.2.3	4. 1, 4. 2, 4. 3	Verde	aVerde	Excepción Evento Ilegal
CI.3.0	1. 1	SinServicio	Encendido	Excepción Evento Ilegal
CI.3.1	1. 1	SinServicio	aAmarillo	Excepción Evento Ilegal
CI.3.2	1. 1	SinServicio	aInt	Excepción Evento Ilegal
CI.3.3	1. 1	SinServicio	Fallo	Excepción Evento Ilegal
CI.3.4	1. 1	SinServicio	aRojo	Excepción Evento Ilegal
CI.3.5	1. 1	SinServicio	aVerde	Excepción Evento Ilegal
CI.4.0	4. 1, 4. 2, 4. 3, 4. 4	Amarillo	Encendido	Excepción Evento Ilegal
CI.4.1	4. 1, 4. 2, 4. 3, 4. 4	Amarillo	aAmarillo	Excepción Evento Ilegal
CI.4.2	4. 1, 4. 2, 4. 3, 4. 4	Amarillo	ResetOK	Excepción Evento Ilegal
CI.4.3	4. 1, 4. 2, 4. 3, 4. 4	Amarillo	aVerde	Excepción Evento Ilegal
CI.5.0	1. 1, 1. 2, 1. 3	Intermitente	Encendido	Excepción Evento Ilegal
CI.5.1	1. 1, 1. 2, 1. 3	Intermitente	aAmarillo	Excepción Evento Ilegal
CI.5.2	1. 1, 1. 2, 1. 3	Intermitente	aInt	Excepción Evento Ilegal
CI.5.3	1. 1, 1. 2, 1. 3	Intermitente	ResetOK	Excepción Evento Ilegal
CI.5.4	1. 1, 1. 2, 1. 3	Intermitente	aVerde	Excepción Evento Ilegal
CI.6.0	1. 1, 1. 2	Rojo	Encendido	Excepción Evento Ilegal
CI.6.1	1. 1, 1. 2	Rojo	aAmarillo	Excepción Evento Ilegal
CI.6.2	1. 1, 1. 2	Rojo	ResetOK	Excepción Evento Ilegal
CI.6.3	1. 1, 1. 2	Rojo	aRojo	Excepción Evento Ilegal

Tabla 5.6 Caminos Ilegales

Diagrama de Estados Jerárquico

El caso de estudio, es esta vez el diagrama de estados jerárquico simplificado de una aguja ferroviaria.

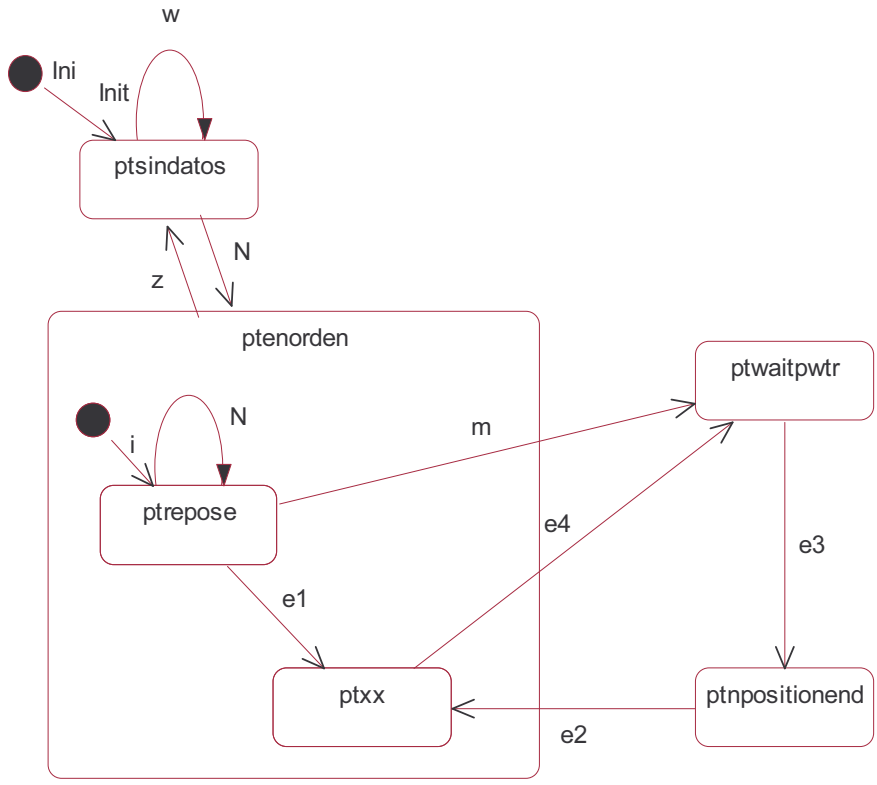


Figura 5.9 Diagrama de Estados

Las tablas de requisitos asociadas son:

estado	requisito
Ini	Inicialización
ptsindatos	R1
ptenorden	R2
ptrepose	R21
ptxx	R22
ptwaitpwtr	R3
ptnpositionend	R4

transición	requisito
W	T11
N	T12
N*	T2int
e1	T2in
M	T23
e4	T23
e3	T34
e2	T42
Init	Inicialización
Z	T21

Tabla 5.7 Requisitos de los estados y de las transiciones

Con estos datos, la herramienta PLOT genera los resultados que aparece a continuación:

		w	N	e1	M	e4	e3	e2	Init	z
ptsindatos	Normal	ptsindatos	ptrepose							
ptrepose	Normal		ptrepose	ptxx	ptwaitpwtr					ptsindatos
ptxx	Normal					ptwaitpwtr				ptsindatos
ptwaitpwtr	Normal						ptnpositionend			
ptnpositionend	Normal							ptxx		
Ini	Inicial								ptsindatos	

Tabla 5.8 Tabla de transiciones

		w	N	e1	m	e4	e3	e2	Init	z
ptsindatos	R1	T11	T12							
ptrepose	R2;R21		T2int	T2in	T23					T21
ptxx	R2;R22					T23				T21
ptwaitpwtr	R3						T34			
ptnpositionend	R4							T42		
Ini	Inicialización								Inicialización	

Tabla 5.9 Tabla de requisitos

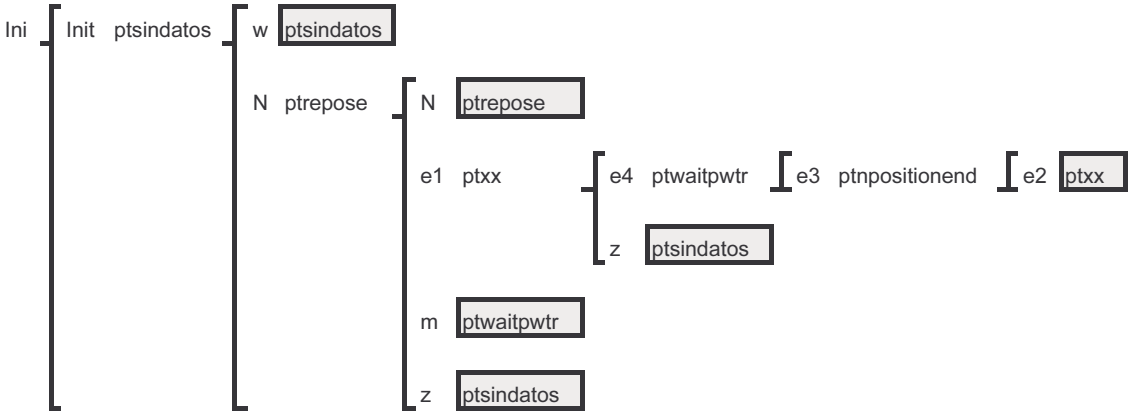


Figura 5.10 Arbol de Estados

ID	Evento	Estado esperado
1. 1	Init	ptsindatos
1. 2	w	ptsindatos
2. 1	Init	ptsindatos
2. 2	N	ptrepose
2. 3	N	ptrepose
3. 1	Init	ptsindatos
3. 2	N	ptrepose
3. 3	e1	ptxx
3. 4	e4	ptwaitpwtr
3. 5	e3	ptnpositionend
3. 6	e2	ptxx

ID	Evento	Estado esperado
4. 1	Init	ptsindatos
4. 2	N	ptrepose
4. 3	e1	ptxx
4. 4	z	ptsindatos
5. 1	Init	ptsindatos
5. 2	N	ptrepose
5. 3	m	ptwaitpwtr
6. 1	Init	ptsindatos
6. 2	N	ptrepose
6. 3	z	ptsindatos

Tabla 5.10 Prueba se recorrido de la máquina de estados

ID	Caso de prueba			Resultado esperado
	Inicialización	Estado	Evento	
CI.1.0	1. 1	ptsindatos	e1	Excepción Evento Ilegal
CI.1.1	1. 1	ptsindatos	M	Excepción Evento Ilegal
CI.1.2	1. 1	ptsindatos	e4	Excepción Evento Ilegal
CI.1.3	1. 1	ptsindatos	e3	Excepción Evento Ilegal
CI.1.4	1. 1	ptsindatos	e2	Excepción Evento Ilegal
CI.1.5	1. 1	ptsindatos	Init	Excepción Evento Ilegal
CI.1.6	1. 1	ptsindatos	Z	Excepción Evento Ilegal
CI.2.0	2. 1, 2. 2	ptrepose	W	Excepción Evento Ilegal
CI.2.1	2. 1, 2. 2	ptrepose	e4	Excepción Evento Ilegal
CI.2.2	2. 1, 2. 2	ptrepose	e3	Excepción Evento Ilegal
CI.2.3	2. 1, 2. 2	ptrepose	e2	Excepción Evento Ilegal
CI.2.4	2. 1, 2. 2	ptrepose	Init	Excepción Evento Ilegal
CI.3.0	3. 1, 3. 2, 3. 3	ptxx	W	Excepción Evento Ilegal
CI.3.1	3. 1, 3. 2, 3. 3	ptxx	N	Excepción Evento Ilegal
CI.3.2	3. 1, 3. 2, 3. 3	ptxx	e1	Excepción Evento Ilegal
CI.3.3	3. 1, 3. 2, 3. 3	ptxx	M	Excepción Evento Ilegal
CI.3.4	3. 1, 3. 2, 3. 3	ptxx	e3	Excepción Evento Ilegal
CI.3.5	3. 1, 3. 2, 3. 3	ptxx	e2	Excepción Evento Ilegal
CI.3.6	3. 1, 3. 2, 3. 3	ptxx	Init	Excepción Evento Ilegal
CI.4.0	3. 1, 3. 2, 3. 3, 3. 4	ptwaitpwtr	W	Excepción Evento Ilegal
CI.4.1	3. 1, 3. 2, 3. 3, 3. 4	ptwaitpwtr	N	Excepción Evento Ilegal
CI.4.2	3. 1, 3. 2, 3. 3, 3. 4	ptwaitpwtr	e1	Excepción Evento Ilegal
CI.4.3	3. 1, 3. 2, 3. 3, 3. 4	ptwaitpwtr	M	Excepción Evento Ilegal

ID	Caso de prueba			Resultado esperado
	Inicialización	Estado	Evento	
CI.4.4	3. 1, 3. 2, 3. 3, 3. 4	Ptwaitpwtr	e4	Excepción Evento Ilegal
CI.4.5	3. 1, 3. 2, 3. 3, 3. 4	Ptwaitpwtr	e2	Excepción Evento Ilegal
CI.4.6	3. 1, 3. 2, 3. 3, 3. 4	Ptwaitpwtr	Init	Excepción Evento Ilegal
CI.4.7	3. 1, 3. 2, 3. 3, 3. 4	Ptwaitpwtr	z	Excepción Evento Ilegal
CI.5.0	3. 1, 3. 2, 3. 3, 3. 4, 3. 5	ptnpositionend	w	Excepción Evento Ilegal
CI.5.1	3. 1, 3. 2, 3. 3, 3. 4, 3. 5	ptnpositionend	N	Excepción Evento Ilegal
CI.5.2	3. 1, 3. 2, 3. 3, 3. 4, 3. 5	ptnpositionend	e1	Excepción Evento Ilegal
CI.5.3	3. 1, 3. 2, 3. 3, 3. 4, 3. 5	ptnpositionend	m	Excepción Evento Ilegal
CI.5.4	3. 1, 3. 2, 3. 3, 3. 4, 3. 5	ptnpositionend	e4	Excepción Evento Ilegal
CI.5.5	3. 1, 3. 2, 3. 3, 3. 4, 3. 5	ptnpositionend	e3	Excepción Evento Ilegal
CI.5.6	3. 1, 3. 2, 3. 3, 3. 4, 3. 5	ptnpositionend	Init	Excepción Evento Ilegal
CI.5.7	3. 1, 3. 2, 3. 3, 3. 4, 3. 5	ptnpositionend	z	Excepción Evento Ilegal
CI.6.0	Estado por defecto	Ini	w	Excepción Evento Ilegal
CI.6.1	Estado por defecto	Ini	N	Excepción Evento Ilegal
CI.6.2	Estado por defecto	Ini	e1	Excepción Evento Ilegal
CI.6.3	Estado por defecto	Ini	m	Excepción Evento Ilegal
CI.6.4	Estado por defecto	Ini	e4	Excepción Evento Ilegal
CI.6.5	Estado por defecto	Ini	e3	Excepción Evento Ilegal
CI.6.6	Estado por defecto	Ini	e2	Excepción Evento Ilegal
CI.6.7	Estado por defecto	Ini	z	Excepción Evento Ilegal

Tabla 5.11 Caminos Ilegales

5.4.4.4 Valoración de los resultados alcanzados

El entorno de herramientas realizadas ha demostrado que la estrategia de generación de pruebas de forma automatizada a partir de diagramas de estados se puede soportar mediante la adaptación de herramientas software comerciales existentes en el mercado. En la medida que haya más recursos disponibles para realizar dicha adaptación, se podrán obtener entornos de prueba con mayores prestaciones.

En cualquier caso, las hojas de cálculo se han revelado como la herramienta base más práctica para escribir en ellas los casos de prueba y aplicar sobre ellas posteriormente las herramientas que generan los casos de prueba ejecutables. Al escribir el caso de prueba, los aspectos de generación de código no se deben de tener en cuenta, sino únicamente los aspectos funcionales. Sobre las hojas de cálculo se pueden integrar diferentes herramientas que tengan funciones diferentes (trazabilidad, medida de cobertura, etc.).

Las herramientas realizadas en el caso de aplicación industrial muestran cómo se puede gestionar un proceso de validación complejo y de gran envergadura mediante el método propuesto en la Tesis Doctoral, si bien se ha aplicado una variante simplificada del mismo, pues no se han utilizado diagramas de estados jerárquicos sino diagramas de estados convencionales (sin jerarquía).

La herramienta PLOT de generación de casos de prueba ha servido para mostrar que se puede hacer más fácil el proceso introduciendo los diagramas de estados en forma gráfica y que se pueden utilizar diferentes estrategias de prueba de diagramas de estados en el marco del método de pruebas propuesto en la Tesis Doctoral, a diferencia del caso de aplicación industrial que ha utilizado por su simplicidad la estrategia de comprobar las transiciones simples de un estado a otro. La herramienta PLOT también ha mostrado cómo se pueden utilizar los diagramas de estados jerárquicos, mejorando respecto a las herramientas del caso de aplicación industrial que sólo utilizan diagramas de estados simples. Aparte de estas dos mejoras, la herramienta PLOT genera los casos de prueba a partir del diagrama de estados de forma automática, a diferencia de las herramientas del caso de aplicación industrial, que requieren que esta tarea se haga manualmente. En este sentido se puede concluir que la herramienta PLOT se ajusta más al ideal de máxima automatización de pruebas posible propio del método de pruebas de la Tesis Doctoral.

La trazabilidad con los requisitos es la parte no automatizable del método de pruebas y requiere grandes conocimientos del dominio. El esfuerzo de aplicarla sobre los requisitos de detalle se justifica plenamente en el caso de un sistema similar al del caso de aplicación industrial, donde el número de requisitos es muy elevado y donde la certificación de seguridad obliga a documentar de forma detallada la cobertura de los requisitos en las pruebas. En otro tipo de sistemas, habría que analizar si este esfuerzo merece la pena o no, en función del grado de detalle de la especificación de requisitos y la complejidad del sistema.

5.5 Otras Herramientas Software Relacionadas

A continuación se hace referencia a otras dos herramientas software relacionadas con las expuestas hasta ahora y que han sido también desarrolladas en el DIT de la UPM.

La primera herramienta [CeR03] está orientada a la gestión de requisitos de Línea de Producto Software y se llama ENAGER (Environment for requirements Analysis and Management). La herramienta estructura la información según el árbol de navegación que se muestra en la figura y que esta basado en el metamodelo de requisitos para Líneas de Producto Software de [CeR03]. ENAGER

es capaz de generar informes de métricas de los requisitos que dan idea de la calidad y el tamaño a nivel global de la especificación, como por ejemplo, el número de requisitos total, el número de requisitos específicos de cada producto, etc. Otra característica importante de la herramienta es la generación de matrices de trazabilidad, donde se muestran las relaciones entre los diferentes tipos elementos del árbol de navegación, como por ejemplo actores y casos de uso.

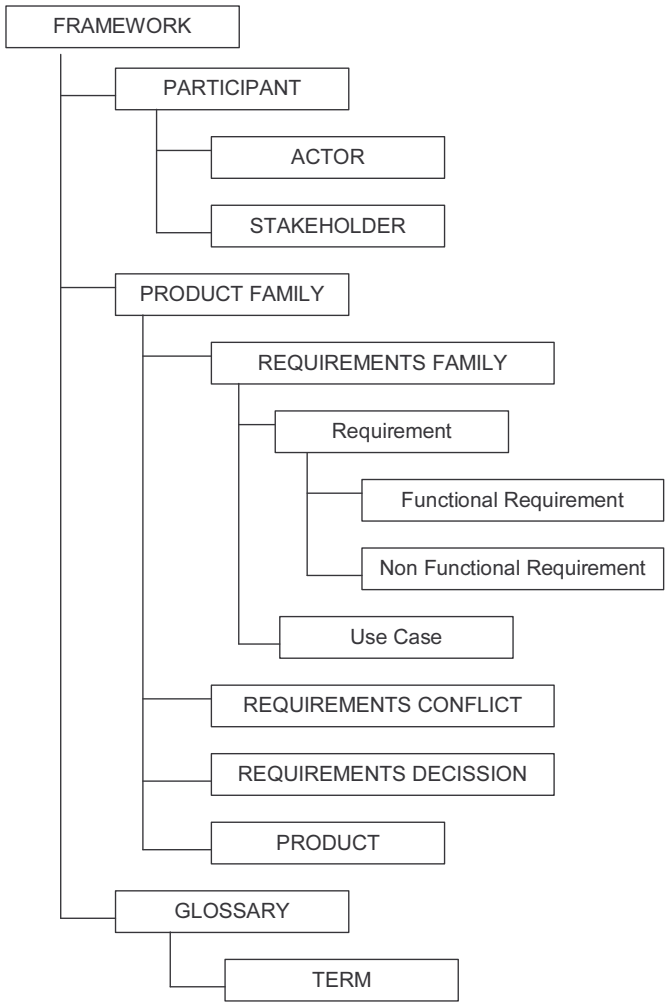


Figura 5.11 Arbol de navegación del usuario de la herramienta ENAGER

La otra herramienta [DuA03][Arc03] sirve para validar arquitecturas software de tiempo real utilizando la técnica RMA (Rate Monotonic Analysis). Al igual que las herramientas del entorno PLAT, esta herramienta exige como condición previa para su aplicación el analizar los requisitos del sistema asignándolos a seis características de calidad (funcionalidad, robustez, eficiencia, usabilidad, mantenibilidad y portabilidad) según lo que indica el estándar [ISO91]. Se elaboran diferentes modelos de la arquitectura con diferentes procesos y se les asignan prioridades en función del análisis de requisitos previo, que la herramienta verifica mediante la técnica RMA, y como resultado final se tiene un veredicto acerca de la posibilidad de satisfacer los requisitos mediante cada uno de los modelos de la arquitectura.

6 Conclusiones

En este capítulo se presenta un resumen de las contribuciones realizadas y las líneas futuras de investigación propuestas en la Tesis Doctoral. Como se dice al principio, esta Tesis Doctoral tiene un marcado carácter industrial, por un lado por el tema elegido y por otro lado por la experiencia profesional del autor, que ha influido en la elección del tema y en su tratamiento. En la industria existe el problema que las Líneas de Producto Software pretenden resolver, que es el de usar sistemas software y hardware genéricos adaptándolos para diferentes proyectos.

En el estudio del Estado de la Ciencia en los temas de Líneas de Producto Software, se obtuvieron las siguientes conclusiones:

- Las Líneas de Producto Software son una incipiente estrategia de desarrollo de software encuadrada dentro de todo un conjunto de técnicas de más o menos reciente aparición orientadas a la mejora de la eficiencia del desarrollo de software por la vía de la reutilización del código y de la arquitectura software y por el uso de componentes e interfaces estandarizados.

- Las Líneas de Producto Software tiene dos grandes áreas, que son la Ingeniería del Dominio, que es donde se enmarca la definición de la arquitectura y de los elementos comunes, y la Ingeniería de Aplicación, que es donde se realiza la obtención de las diferentes variantes o productos a partir de los elementos genéricos, también llamados “core assets” o activos básicos en la literatura [Du01][ClNo01][CA03].

- Los beneficios de las Líneas de Producto Software son la mejora de la eficiencia y la disminución del esfuerzo, existiendo ejemplos industriales que demuestran su utilidad en la práctica. Estas ventajas se obtienen a costa del mayor esfuerzo inicial de delimitar e implementar los activos básicos o partes genéricas, y de diseñar el mecanismo de obtención de las diferentes variantes a partir de dichos activos básicos.

- Implantar la técnica de Líneas de Producto supone cambios en la estructura de la organización que realiza el desarrollo y en todas las fases del ciclo de vida del software, incluyendo las pruebas.

- Las diferentes técnicas usadas para la especificación de requisitos de Líneas de Producto Software estructuran los requisitos en forma jerárquica o de árbol, colocando en la base del árbol los requisitos comunes y en las ramas los requisitos específicos de cada producto. Entre todas estas técnicas destaca por su grado de detalle FODA (Feature Oriented Domain Analysis), pues no sólo permite representar jerarquías sino también relaciones de dependencia o exclusión entre las diferentes características variantes de la Línea de Producto Software.

- La mayoría de las herramientas software comerciales actuales no están orientadas explícitamente al desarrollo de Líneas de Producto Software y precisan de una adaptación a medida para ello.

- Aunque no hay un lenguaje de programación específico de Líneas de Producto Software, los lenguajes orientados a objetos [Boo96], y los patrones de diseño [GHJV95] se emplean frecuentemente en las Líneas de Producto Software, pues permiten modelar de forma cómoda y elegante, por ejemplo mediante la herencia, la variabilidad de los diferentes productos. En la fase de análisis de Líneas de Producto Software se usa habitualmente UML, por el mismo motivo.

- Las pruebas de una Línea de Producto Software implican el desarrollo de una infraestructura de prueba común para todos sus componentes. Dependiendo del sistema concreto, puede ser posible probar varios productos simultáneamente probando la parte común y si esto no es posible, se pueden reutilizar los elementos de prueba comunes (casos de prueba, entorno, proceso de prueba, etc.).

- Las Líneas de Producto Software, al igual que cualquier otra técnica, no son el remedio universal para cualquier tipo de problema. La gestión del desarrollo es más compleja con las Líneas de Producto Software y está sujeta a mayores riesgos que las de los desarrollos realizados de forma individual, y por eso, antes de tomar la decisión de implantar una Línea de Producto Software, hay que evaluar desde el punto de vista técnico y económico si el esfuerzo de hacerlo merece la pena [Sch02].

Puesto que el tema de la Tesis Doctoral es el de las pruebas de software, dentro de las Líneas de Producto Software, se ha estudiado también en la parte de Estado de la Ciencia el proceso de pruebas del software y sus diferentes fases, con los siguientes resultados:

- El modelo de ciclo de vida en cascada en el que las pruebas son la fase final se ha sustituido por una sucesión de pequeños de ciclos de vida iterativos (minúsculos en el caso de la técnica de Extreme Programming) con fases de prueba al final de cada iteración.

- La utilización de modelos formales o semiformales, como las máquinas de estados, permite elaborar una especificación del comportamiento de la implementación detallada y no ambigua, que luego puede usarse para las pruebas.

- Hay múltiples estrategias de prueba de un diagrama de estados, y se debe de utilizar aquella que optimice el esfuerzo de pruebas una vez que estén claros los objetivos de cobertura, es decir hasta dónde se quiere probar. Dentro de las descritas en la literatura, la estrategia de pruebas N+ [Bin00] se ha elegido como un compromiso adecuado entre simplicidad y exhaustividad.

- Los modelos de UML son intuitivos y tiene gran capacidad expresiva, si bien no existe aún una técnica definida y clara de generación de pruebas a partir de dichos modelos, como en el caso del lenguaje SDL [SDL95] y su notación de pruebas asociada TTCN [ITU92]. Un esfuerzo significativo para subsanar esta carencia es el UML Testing Profile.

- Los diagramas de estados jerárquicos o statecharts de UML tienen más capacidad expresiva que las máquinas de estados finitos de Moore y Mealy y son una técnica adecuada para modelar el comportamiento dinámico de un objeto que reacciona a los estímulos recibidos del exterior, usándose ampliamente en el análisis y diseño de sistemas software de tiempo real [HaPo98].

- Los escenarios, conocidos también como diagramas de secuencia de UML son una técnica adecuada para representar en el tiempo la sucesión de envío de mensajes entre los diferentes elementos del sistema y también se usan habitualmente en el análisis y diseño de sistemas software de tiempo real [Dou99].

- Se está aumentando progresivamente la eficiencia y el grado de exhaustividad de las pruebas del software, mediante el uso de herramientas comerciales o de libre distribución software específicas para automatización de pruebas [Few99] y la definición de procesos estructurados de prueba.

Una vez presentado el Estado de la Ciencia, se han expuesto los resultados del trabajo realizado, empezando por la definición de un método de pruebas para Líneas de Producto Software de Tiempo Real cuyas características son:

- *Modelo de ciclo de vida del software*, basado en el modelo en V clásico, al que se le añade por un lado una fase previa de análisis del dominio, en la que se diferencian los requisitos genéricos de los específicos y se elabora una lista con las características de cada variante, y por otro lado se detallan dos aspectos específicos de las Líneas de Producto Software en la fase de pruebas: el mecanismo de generación de las variantes y las pruebas de las variantes en las interfaces externas. Este modelo de ciclo de vida puede aplicarse también en el marco de un proceso de desarrollo iterativo, si en las iteraciones se añaden nuevos requisitos al sistema. Dentro de todas las fases del ciclo de vida, el ámbito del método es la fase de pruebas de sistema también denominadas de pruebas de validación, puesto que en dichas pruebas es donde se verifica que la implementación satisface los requisitos. Se asume que no se va a poder probar de una vez todos los elementos de la Línea de Producto Software, sino que se van a tener que repetir las pruebas de los requisitos comunes para todos los productos, aparte de realizar las pruebas de los requisitos específicos con cada uno de ellos, y por ello se define la automatización de las pruebas como un elemento esencial del método para optimizar el esfuerzo de pruebas.

- *Concepto de Conformidad.* Conformidad significa que la implementación se ajusta a su especificación. Se aplica la definición de conformidad propuesta por [ITU97], basada en la adecuación de la implementación respecto de un modelo formal de la especificación, que en el caso del método de pruebas es un diagrama de estados jerárquico. Es decir, se modelan los requisitos mediante diagramas de estados jerárquicos y asumiendo (postulado de prueba o *test assumption*) que este modelo de estados jerárquicos refleja adecuadamente el comportamiento especificado para la implementación, se prueba esa implementación utilizando esos diagramas de estados.

- *Concepto de Analogía.* Es como se denomina la relación entre los miembros de la Línea de Producto fundamentada en la semejanza parcial que existe entre ellos. Se ha distinguido en los requisitos entre requisitos de carácter global o características (*features*) y requisitos detallados. Se ha elaborado una definición formal de la analogía en el nivel de las características, pues en el nivel de los requisitos detallados no se ha visto que tenga sentido, pues lo habitual es que el número de requisitos detallados sea elevado y es más práctico un criterio de tipo cuantitativo como definir que los miembros de una Línea de Producto Software deben de tener el ochenta por ciento de sus requisitos detallados en común. Para establecer la analogía a partir de la matriz de productos y características denominada como *mapa de producto* se define la noción de *concepto* como una combinación de productos y características tales que todos los productos tienen todas las características y todas las características pertenecen a todos los productos. Los miembros de la Línea de Producto Software son *análogos* cuando existe un *concepto* general para todos ellos distinto del conjunto vacío.

- *Concepto de Trazabilidad*. Hay una correspondencia entre Requisitos y Casos de Prueba, de forma que cada caso de prueba tiene sus requisitos asociados y viceversa, con el objetivo de que todos los requisitos se verifiquen. Existen varios tipos de trazabilidad, nombradas por la posición de las fases en el modelo clásico en V del ciclo de vida software: por un lado está la *trazabilidad vertical* entre casos de prueba y elementos de la arquitectura (qué partes de la arquitectura se han probado), que no es un objetivo prioritario del método de pruebas, y por otro la *trazabilidad horizontal* entre requisitos y casos de prueba, que sí que es importante en el método, y cuya cobertura de pruebas se mide conociendo cuantos requisitos se han probado respecto de todos los requisitos en total. También existe un segundo nivel de *trazabilidad horizontal* constituido por los casos de prueba de integración, los casos de pruebas unitarias y la arquitectura del sistema, que no es prioritario en el método y para el que se definen también unas métricas de cobertura.

- *Definición de un modelo de Requisitos de Línea de Producto Software*. Como consecuencia del carácter prioritario de la trazabilidad entre requisitos y casos de prueba, se ha elaborado un modelo

de requisitos de Línea de Producto Software de Tiempo Real. Se han establecido dos niveles de detalle en los requisitos, que son el nivel de requisitos globales o de características y el nivel de requisitos de detalle. El nivel de requisitos globales está descrito por la lista de características o *features* de la Línea de Producto Software y el nivel detallado está descrito por la Especificación de Requisitos de Sistema. Teniendo en cuenta las necesidades del método de pruebas, se ha confeccionado un metamodelo del nivel de características, en el cual se distingue entre características genéricas y características específicas, y se detallan los roles de responsable y de cliente de la Línea de Producto Software. Esa misma distinción entre genérico y específico se ha establecido en el nivel de requisitos de detalle, donde se ha confeccionado también otro metamodelo de Requisito de Línea de Producto Software, que se ha extendido para requisitos de tiempo real añadiendo al requisito una *función de utilidad* en los casos que sea necesario definir de forma más precisa el plazo crítico especificado por el requisito.

- *Organización de los elementos de Prueba conforme al modelo de Requisitos.* Se analiza la especificación de requisitos y se elaboran modelos de diagramas de estados de la funcionalidad especificada por los requisitos, asociando requisitos a las transiciones y los estados del diagrama. Todos los requisitos que parecen en el diagrama constituyen una *clase de prueba.* A partir de este modelo se generan *casos de prueba*, que pueden ser simples como por ejemplo, recorrer una transición de un estado a otro, o algo más complejos, como recorrer en secuencia varios estados y varias transiciones. La implementación del caso de prueba se denomina *dato de prueba*. Todos estos elementos de prueba se han definido también de una manera formal y se ha propuesto una *notación de prueba* para expresarlos.

- *Uso de diagramas de Estados Jerárquicos o statecharts*. Para considerar las variaciones en los requisitos propias de las Líneas de Producto Software, se ha diferenciado entre diagrama de estados genérico y específico, que extiende el diagrama genérico añadiéndole nuevos estados o transiciones. Para modelar los requisitos de tiempo real con los diagramas de estados, se usan los eventos de *timeout* definidos en [HaPo98].

- *Uso de Escenarios*. Para probar las interacciones entre objetos pertenecientes a clases de prueba diferentes, se utilizan los modelos de escenarios o diagramas de interacción aplicando la misma filosofía de trazabilidad que para las máquinas de estados, es decir, se asocian requisitos a cada interacción del escenario y se especifican casos de prueba que pueden abarcar una o varias interacciones.

- *Estrategia de Reducción del Conjunto de Pruebas*. Con el fin de optimizar los recursos y el tiempo disponibles para las pruebas, se ha definido un *modelo de fallos*, consistente en una lista de posibles errores de la implementación respecto de la especificación del diagrama de estados, que se describen en lenguaje natural y de modo formal. La estrategia de reducción de pruebas es una opción pragmática y necesita ser validada en cada caso práctico concreto. No se han establecido prioridades en los casos de prueba, si bien se han propuesto algunos posibles criterios para hacerlo.

- *Funciones de Cobertura y de Coste de las Pruebas*. Se han propuesto sendas funciones para la medición de la exhaustividad y del coste las pruebas, que se pueden usar para el seguimiento de las pruebas, teniendo en cuenta sus limitaciones. Por un lado, la cobertura se mide respecto del número total de casos de prueba una vez que se ha aplicado la estrategia de reducción de pruebas, y por el otro, los cálculos realizados mediante la función de coste necesitan ser contrastados con datos reales.

- *Arquitectura de Pruebas y Puntos de Control y Observación*. Conforme al modelo abstracto de arquitectura de pruebas de [ITU97] se ha definido la información que debe de facilitar un punto de control y observación (PCO), de acuerdo con la definición formal de los casos de prueba y las

características de los modelos de diagramas de estados jerárquicos utilizados para modelar la especificación de requisitos. Lo deseable es que para todos los productos se pueda utilizar los mismos PCO e IAP (Puntos de Acceso a la Implementación), y en caso de no ser así, habría que distinguir entre puntos de control y acceso genéricos o específicos.

- *Metamodelo de elementos de prueba.* Se ha propuesto un metamodelo en notación UML donde se representan todos los elementos del método de prueba y sus relaciones. Esta es una de las contribuciones más relevantes de la Tesis Doctoral, ya que se muestra cómo se han articulado las relaciones entre requisitos y casos de prueba por medio de los modelos de diagramas de estados jerárquicos y de escenarios, y cómo repercute la variabilidad de la Línea de Producto Software en esas relaciones.

- *Impacto en la mejora del proceso software*. Las áreas de mejora del proceso software afectadas por el método son la gestión de requisitos, el seguimiento y la predicción del proyecto software y la ingeniería de Producto software, y los niveles CMM [SEI91] relevantes para el método son el 2 y el 3. Aplicando la nueva versión de CMM, el CMMI, se pueden definir áreas de mejora diferenciadas para la ingeniería de dominio y la ingeniería de aplicación de Líneas de Producto Software.

- *Flujo de Actividades*. Se ha fijado la secuencia de actividades necesarias incluyendo las relativas a la automatización de las pruebas para aplicar el método tanto en un producto individual como en una Línea de Producto Software, delimitando el ámbito de aplicación preferente de las posibles herramientas software.

Con el fin de brindarle soporte a la metodología de pruebas descrita, se ha desarrollado en el marco de un proyecto de fin de carrera una herramienta software para la generación de casos de prueba a partir de un diagrama de estados jerárquico anotado con información de requisitos según la estrategia de pruebas N+ [Bin 00]. La herramienta contempla sólo los elementos más básicos de la notación de UML para diagramas de estados (transiciones, eventos, estados y jerarquías de estados), pues no se pretendía realizar una herramienta de carácter comercial, sino un prototipo que pudiera mostrara en la práctica la aplicación del método de pruebas. Aparte de dicha herramienta, se ha colaborado en el diseño de otras herramientas específicas de generación de datos de prueba, automatización de pruebas y generación de veredictos de prueba para el caso práctico industrial que ha servido para validar el método de pruebas. La herramienta desarrollada está enmarcada al área de investigación sobre Líneas de Producto Software del DIT de la UPM donde han surgido otras herramientas software relacionadas, a las cuales se ha hecho referencia en la Tesis Doctoral.

Tanto los aspectos metodológicos como las herramientas que se desarrollaron como soporte fueron aplicados a un caso de estudio industrial real en el cual pudo evaluarse el cumplimiento de los objetivos establecidos, y cuya experiencia ha servido para refinar y extender la parte metodológica de la Tesis Doctoral:

- El dominio de aplicación elegido ha sido el de los sistemas de control del tráfico ferroviario, donde el autor acumula una experiencia de trabajo de diez años en distintas tareas relacionadas con el desarrollo y las pruebas del software. Concretamente se ha aplicado el método propuesto en la Tesis Doctoral en la validación de una línea de Productos de tiempo real de enclavamientos electrónicos, que son sistemas encargados de garantizar la integridad en los movimientos de los trenes dentro de una estación mediante el mando y supervisión de las agujas, señales y demás dispositivos relevantes para la seguridad.

- Con el objetivo de aclarar con mayor exactitud el alcance del caso práctico, se ha incluido una parte de introducción previa en la Tesis Doctoral explicando el dominio de aplicación y las

características especiales de los procesos de validación y certificación a los que están sujetos los enclavamientos electrónicos y los sistemas ferroviarios en general.

- Se ha descrito la parte genérica y las fuentes de variabilidad de la Línea de Producto Software, así como una somera descripción de los mecanismos de generación de las diferentes variantes.

- Asimismo se ha explicado cómo se aplica a la Línea de Producto Software el modelo de ciclo de vida impuesto por el estándar de seguridad CENELEC [CEN97] para el desarrollo de sistemas de control ferroviarios, entre cuyos principales requisitos está el de la trazabilidad entre requisitos y pruebas, con su medición de cobertura de pruebas correspondiente.

- Se ha trabajado utilizando fundamentalmente diagramas de estados jerárquicos, y por eso no se ha podido contar con resultados experimentales para validar el método en lo referente a la prueba de escenarios.

- Se ha relatado también cómo el método de pruebas se ha aplicado para la validación según la norma CENELEC sobre dos subsistemas con requisitos de seguridad pertenecientes al enclavamiento electrónico que también están sujetos a una cierta variabilidad indicando los orígenes de dicha variabilidad y cómo se han desarrollado dos entornos de validación automática uno tipo *batch* y otro interactivo, en los que hay trazabilidad entre los casos de prueba y los requisitos, aportándose un balance de los resultados prácticos alcanzados.

La formalización del método ha sido posterior a la realización del caso de aplicación práctica, cuando ya se contaba con la perspectiva suficiente para conocer los pros y los contras de la aplicación del método y se ha podido cotejar con mayor conocimiento de causa con otros métodos de prueba y otras experiencias industriales existentes en la literatura.

En resumen, la *tesis* planteada y que se ha pretendido demostrar en el presente trabajo es la siguiente:

Es posible mejorar la eficiencia de las pruebas de sistema de las Líneas de Producto Software garantizando la trazabilidad entre requisitos y pruebas y verificando los requisitos de tiempo real usando las siguientes técnicas:

- *Análisis del dominio para elaborar una lista de características de las diferentes variantes y clasificar los Requisitos en Genéricos y Específicos.*

- *Modelado de los requisitos mediante diagramas de estados jerárquicos, incluyendo en los estados y las transiciones información de sus requisitos asociados.*

- *Implantación de un proceso de especificación de pruebas, implementación de pruebas y ejecución de pruebas automatizada a partir de los modelos de diagramas de estados jerárquicos, que se aplica sobre cada una de las variantes de la Línea de Producto Software.*

Además del cumplimiento de los objetivos básicos de la Tesis Doctoral, se han realizado otras aportaciones adicionales:

- Se ha realizado en primer lugar unas directivas de uso de los diagramas de estados jerárquicos para facilitar su legibilidad y corrección cuando se usan en el contexto del método de pruebas.

- En la parte de aplicación práctica se ha realizado una síntesis del estado actual de la tecnología de sistemas de señalización ferroviaria y se ha clasificado los tipos de sistemas existentes en función de su relevancia para la seguridad y la explotación del tráfico ferroviario.

- Asimismo se han incluido en la parte de aplicación práctica aspectos relevantes para la prueba de Líneas de Producto Software en el campo de la señalización ferroviaria como las normativas de validación y certificación y las iniciativas europeas de estandarización actuales, especialmente ETCS/ERTMS.

La realización del caso de validación industrial en la compañía Alcatel ha motivado el participar como ponente en el foro industrial sobre herramientas de prueba ICSTEST en Düsseldorf (Alemania) en el año 2002 [EMM02], y de realizar varias publicaciones, a saber, un artículo en la revista especializada en tecnología ferroviaria "*Signal und Draht*" [MeET02], dos "papers" que se publicaron con ocasión de dos workshops sobre Líneas de Producto Software celebrados en Las Palmas y Bilbao respectivamente [MeDu01] y [MeDu00], y otra contribución para un workshop sobre MDA (Model Driven Architecture) [DM04].

6.1 Líneas de Investigación abiertas

Como resultado de la Tesis Doctoral se proponen las líneas de investigación siguientes:

- Profundización en la aplicación del método de pruebas sobre escenarios o diagramas de secuencia de UML. Como se ha dicho ya, esta parte de la aportación metodológica de la Tesis Doctoral no está suficientemente contrastada con la práctica al no haberse podido validar en el caso de aplicación industrial, puesto que entonces se concentró el esfuerzo de automatización de pruebas en los modelos de diagramas de estados y se hizo una validación manual de los escenarios definidos. Habría que analizar si el método propuesto es factible para las pruebas de escenarios y ver qué problemas prácticos surgen en la automatización de los entornos de prueba, que podrían dar lugar a extender la arquitectura de pruebas y la definición de los puntos de control y observación. Otra cuestión es qué criterios se siguen para la selección de escenarios para las pruebas si existen escenarios alternativos y de gestión de errores para un determinado escenario. Una alternativa que podría explorarse es la conversión de los escenarios en diagramas de estados y a partir de ahí aplicarles el mismo proceso, o explorar las posibilidades nuevas que ofrece UML 2 en los escenarios para especificar de forma completa las interacciones del sistema estableciendo jerarquías de escenarios.

- Introducción de un mayor grado de formalismo en los diagramas de estados, manteniendo el mismo concepto de trazabilidad entre requisitos y casos de prueba pero expresando los statecharts mediante un lenguaje de carácter más formal que UML y ver si eso proporciona una mayor automatización en la generación de casos de prueba. Como primera opción a evaluar sería OCL, el lenguaje formal asociado a UML propuesto por el OMG, y también se debería de considerar la opción de SDL, como lenguaje de uso industrial con una notación de pruebas asociada (TTCN).

- Combinación del método propuesto con técnicas de verificación automática o *model checking* para detectar fallos en los modelos de diagramas de estados, lo que puede resultar útil si los diagramas

de estados que se manejan son muy complejos y no basta la inspección visual para localizar los errores.

- Estudio de los resultados obtenidos si se cambia la estrategia de pruebas de máquinas de estados. Se ha tomado una estrategia de prueba de máquina de estados de la literatura, y se ha integrado en el método de pruebas, pero podría buscarse otra estrategia de pruebas diferente, de mayor o menor potencia, y ver qué grado de fiabilidad se obtiene en las pruebas.

- Continuación del trabajo en herramientas de soporte de pruebas de Líneas de Producto Software, haciendo hincapié más que en crear herramientas completamente nuevas, en buscar cómo integrar el soporte a Líneas de Producto Software en herramientas de gestión de requisitos y de automatización de pruebas que se estén usando ya en la organización correspondiente, de tal forma que el cambio asociado al usar esta técnica se haga con menor esfuerzo, porque se siguen utilizando herramientas ya conocidas. Lógicamente, a la hora de elegir la herramienta con la que se va a trabajar, debe valorarse un conjunto de factores como grado de extensión en el mercado, soporte disponible, coste de la licencia o si es de libre distribución, etc. Con el fin de que los resultados de las herramientas sean utilizables por otras herramientas, sería importante que se generaran en XMI, el formato estándar para modelos UML, propuesto por el OMG. Para la arquitectura del entorno de pruebas, una base de trabajo interesante es el metamodelo del UML Testing Profile.

7 Glosario

Activos Básicos	Los elementos software genérico de la Línea de Productos Software.
Aguja	Elemento móvil que permite a los trenes cambiar de una vía a otra.
Circuito de vía	Sensor para detectar la presencia de un tren en una sección de vía determinada.
Enclavamiento	Sistema de control ferroviario responsable de la separación en los movimientos de los trenes dentro de una estación.
Itinerario	Ruta reservada y asegurada por el enclavamiento para que circule por ella un tren
Líneas de Producto Software	Conjunto de sistemas software con arquitectura similar y un mecanismo común de generación.
Pruebas de Aceptación	Pruebas realizadas en el entorno final del sistema conjuntamente con el cliente y que habitualmente tienen valor contractual.
Pruebas de Integración	Prueba de las interfaces de los distintos componentes de un sistema
Pruebas de Sistema	Comprobar que el sistema funciona de acuerdo a sus requisitos. También se llaman pruebas de caja negra.
Pruebas Unitarias	Prueba de un módulo software aislado del resto, para lo cual es necesario un cierto conocimiento de su estructura interna (pruebas de caja blanca).
Señal	Indicador, normalmente luminoso, por medio del cual el maquinista conoce la velocidad a la que debe de circular. Los semáforos de los trenes.
Statechart	Diagrama de estados jerárquico
Telemando	Sistema de control ferroviario responsable de optimizar el tráfico a lo largo de una línea férrea determinada.
Trazabilidad	Posibilidad de relacionar los requisitos con las pruebas (trazabilidad horizontal) o con los elementos software de implementación y diseño (trazabilidad vertical)

Validación	Ver Pruebas de Sistema.
Verificación	Comprobar que los resultados (código, documentos, etc.) de una fase del ciclo de vida software existen y son correctos.

8 Abreviaturas

ADD	Architectural Design Document
ASCII	American Standard Code for Information Exchange
ATP	Automatic Train Protection
AVE	Alta Velocidad Española
CAFE	From Concept to Application in System Family Engineering
CAT	Conducción Asistida de Trenes
CENELEC	Comite Europeen de Normalisation Electrotechnique
COTS	Commercial Off The Shelf
CPU	Central Processing Unit
CTC	Centro de Tráfico Centralizado
DDD	Detailed Design Document
DIT	Departamento de Ingeniería de Sistemas Telemáticos
DS	Diagnostic System
EC	Element Controller
ERTMS	European Train Management System
ESAPS	Engineering Software Architectures, Processes and Platforms for System-Families
ETCS	European Control System
FAI	Formación Automática de Itinerarios
FEC	Field Element Controller
FMEA	Fault Modes and Effects Analysis
FODA	Feature Oriented domain Analysis
FREE	Flattened Regular Expression (Expresión regular plana)
GNU	GNU's Not Unix
GSM-R	Global System for Mobile Communications – Railway
HTML	Hypertext Mark Up Language
IC	Interface Controller
ICSTEST	International Conference on Software Testing
IEEE	Institute of Electrical and Electronic Engineers
IESE	Institute for Experimental Software Engineering
IM	Interlocking Module

ISO	International Standards Organisation
ITU	International Telecommunications Union
LSC	Live Sequence Chart
LZB	Linien Zugbeeinflussung (Conducción de Tren por Línea)
MSC	Message Sequence Chart
MTBF	Mean Time Between Failures
OM	Operating Module
OMG	Object Management Group
OMT	Object Modelling Technique
OOA/D	Object Oriented Analysis and Design (Análisis y Diseño Orientado a Objetos)
PC	Personal Computer
PDF	Printable Data Format
PLAT	Product Line Automated Testing
PLOR	Product Line Oriented Requirements Coverage
PLOT	Product Line Oriented Test Builder
POSIX	Portable Operating System Interface
PuLSE	Product Line Software Engineering
RBC	Radio Block Center
RENFE	Red Nacional de Ferrocarriles Españoles
ROI	Return of Investment
ROOM	Real Time Objet Oriented Modelling
RSML	Requirements State Machine Language
RTF	Rich Text Format
RUP	Rational Unified Process
SDL	Specification and Description Language
SEI	Software Engineering Institute
SQL	Standard Query Language
SRS	System Requirements Specification
TAS	Transport Automation Solutions
TTCN	Tree and Tabular Cobined Notation
UML	Unified Modeling Language (Lenguaje Unificado de Modelado)
UPM	Universidad Politecnica de Madrid
VFSM	Virtual Finite State Machine
XML	Extensible Markup Language
XP	Extreme Programming

9 Referencias

[AKZ96] Awad, M., Kuusela, J. Ziegler, J. (1996) *Object Oriented Technology for real-Time Systems. A practical Approach using OMT and Fusion*. New Jersey, Prentice Hall

[Al00] Alagar, V.S., Zheng, M. (2000), *A Rigorous Method for testing Real- Time Reactive Systems*. Paper

[Alh99] Alhir, S. (1998) *UML in a Nutshell*, Prentice Hall, Englewood Cliffs, New Jersey l

[Alo98] Alonso, A., García-Valls, M., de la Puente, J. A. (1998*) Assesment of Timing properties of Family Products*. Second International ESPRIT ARES Workshop, Las Palmas de Gran Canaria, Spain, 1998, Proceedings. Springer Lecture Notes in Computer Science 1429

[Alu00] Alur, R., Etesami, K., Yannakis, M., (2000) *Inference of Message Sequence Charts* Proceedings of the International Conference on Software Engineering ICSE 2000, pp. 304-313

[Amb03] Ambler, S. (2003) *The Elements of UML Style* Cambridge University Press. Cambrige, Inglaterra

[Ame00] America, P., van Wijgerden, J. (2000) *Requirements Modelling for families of Complex Systems* Third International Workshop on Software Architectures for Product Families IWSAPF-3, Las Palmas, Spain. Springer Lecture Notes in Computer Science 1951

[Arc03] Arciniegas, J. L., Cerón, R., Ruiz, J. L., Martínez, V., Dueñas, J. C., (2003) *A case study of Performance analysis for Real-Time Systems* Departamento de Ingeniería de Sistemas Telemáticos de la Universidad Politécnica de Madrid, IASTED, Innsbruck, Austria.

[Ard96] Ardis, M. A., Chaves, J. A. , Jategaonkar, L., Mataga, P., Puchol, C., Staskauskas, M. G., Von Olnhausen, J., *A Framework for Evaluating Specification Methods for Reactive Systems: Experience Report.* IEEE Transactions On Software Engineering, Vol. 22, No. 6: Junio 1996, pp. 378-389

[Arm01] Armstrong, C., (2001) *E-Business Test Modelling with UML*, Cutter IT Journal, Septiembre 2001, Vol 14, No 9, pp 20-28

[Aw97] Awad, M. (1997) *A practical Method and Techniques for Developing Object-Oriented Software for real Time systems*. Helsinki University of Technology. (tesis doctoral)

[Bai95] Bailey, C. (1995) *European Railway Signalling*. Institution of Railway Signalling Engineers. A&C Black, Londres.

[Bal98] Balzer, R. (1998) *An architectural Infrastructure for Product Families*. Second International ESPRIT ARES Workshop, Las Palmas de Gran Canaria, Spain, 1998, Proceedings. Springer Lecture Notes in Computer Science 1429

[Ban00] Bandinelli, S., Sagarduy, G. (2000) *Domain Potential Analysis: Calling the Attention on Business Issues of Product-Lines* Third International Workshop on Software Architectures for Product Families IWSAPF-3, Las Palmas, Spain. pp. 76-81

[Ban01] Bandinelli, S., (2001) *Light Weight Product Family Engineering,* Proceedings of the 4th International Workshop on Product Family Engineering, Springer Lecture Notes in Computer Science 2290

[Bay00] Bayer, J., Flege, O., Gacek, C. (2000) *Creating product Line Architectures* Third International Workshop on Software Architectures for Product Families IWSAPF-3, Las Palmas, Spain. Springer Lecture Notes in Computer Science 1951

[Bay01] Bayer, J., Widen, T., (2001) *Introducing Traceability to Product Lines*, Proceedings of the 4th International Workshop on Product Family Engineering, Springer Lecture Notes in Computer Science 2290

[Bec99] Beck, K., (1999) *Embracing Change with Extreme Programming,*IEEE Computer, Octubre 1999, pp. 70-77

[Bei95] Beizer, B. (1995) *Black-Box Testing. Techniques for functional Testing of Software and systems*, Nueva York, Wiley & Sons

[BGK98] Büssow, R, Geisler R., Klar, M.:(1998) *Specifying Safety-Critical Embedded Systems with Statecharts and Z*: *A Case Study.* FASE 1998 71-87

[Bin00] Binder, R. (2000) *Testing Object-Oriented Systems. Models, Patterns and Tools* Massachusetts: Addison Wesley

[BIN97] Bowen, J., Isaksen, U. y Nissanke, N., (1997) *System and Software Safety in Critical Systems*. Reading, Inglaterra: Reading University, Computer Science Department Technical Report RUCS/97/TR/062/A.

[Bjö00] Björkander, M. (2001) *Graphical Programming using UML and SDL.* IEEE Computer, Enero 2000, pp. 30-35

[Bo96] Boehm, B. (1996) *A Spiral Model of Software Development and enhancement Software Engineering* Notes vol 11(4), p. 22

[Boe00] Böhm, P., Janle, J. (2000) *Das ESTW L90 5 auf der Strecke Kouvola – Pieksämäki in Finnland The ESTW L90 5 on the Line Kouvola – Pieksämäki in Finland*.Signal+Draht 06/2000.

[BoGr96] Bourgoyne, D., Gresham J. (1996) *Object Oriented Modelling for Embedded Print Engine Control*, Objectime Ltd. (paper).

[Boo96] Booch, G. (1996). *Análisis y Diseño orientado a objetos con aplicaciones*, 2ª edición. Massachusetts:Addison-Wesley / Díaz de Santos.

[Br03] Braband, J., Hirao, Y., Luedeke, J. F., (2003) *The relationship between the CENELEC Railway Signalling Standards and other Safety Standards*, Signal und Draht, 12/2003, pp. 32-38

[Br95] Brooks, F., (1995) *The Mythical Man-Month* Edicion 20 Aniversario, Addison-Wesley, Massachussets.

[Bre98] Brehmke, B., (1998) *CENELEC konforme Software Entwicklung bei Siemens Verkehrstechnik,* Signal+Draht, Julio 1998, pp. 5-9

[Buw99] Buwalda, H. Kasdorp, M., (2002) *Getting Automated Testing Under Control* Software Testing & Quality Engineering pp. 39-44

[CA03] CAFE Project (2003) *Technology Validation* Consortium wide deliverable. WP4

[CaDu01] Capilla, R. Dueñas, J. C. (2001) *Modelling Variability with Features in Distributed Architectures* Proceedings of the 4th International Workshop on Product Family Engineering, Springer Lecture Notes in Computer Science 2290

[CaDu03] Capilla, R. Dueñas, J. C. (2003). *Light-Weight Product-Lines for Evolution and Maintenance of Web Site.* 7th European Conference on Software Maintenance and Reengineering (CSMR 2003), 26-28 March 2003, Benevento, Italy, Proceedings. IEEE Computer Society 2003

[CAF01] CAFE (2001) *CAFE Framework V 0.1*, Project Public Results, accesibles en www.esi.es

[Ce01] Cerón, R. , Dueñas, J. C., (2001) *UML based techniques* Technical Report UPM-WP2-0010-1, ESAPS Project, 2000. Resultados públicos del proyecto disponibles en www.esi.es

[CeA03] Cerón, R., Arciniegas, J. L., Ruiz, J. L., Dueñas, J. C., Capilla, R.(2003) *Architectural Modelling in Product Family Context,* Departamento de Ingeniería de Sistemas Telemáticos de la Universidad Politécnica de Madrid, Paper.

[CEN97] Comité Europeén de Normalisation Électrotechnique CENELEC (1997), prEN 50128, Railway applications. Software for railway control and protection systems.

[Cer00] Cerón, R., Dueñas, J. C., de la Puente, J. A., (2000) *A first Assessment of development process with respect to product lines and component based development*. Third International Workshop on Software Architectures for Product Families IWSAPF-3, Las Palmas, Spain. Springer Lecture Notes in Computer Science 1951

[CeR03] Cerón, R., Ruiz, J. L., Valsera, F., Arciniegas, J. L., Dueñas, J. C., (2003) *A Meta model and a Tool for System Family Requirements Engineering,* Departamento de Ingeniería de Sistemas Telemáticos de la Universidad Politécnica de Madrid, Informe técnico.

[CeV03] Cerón, R., Valsera, F., Arciniegas, J. L., Ruiz, J. L., Dueñas, J. C., (2003) *A Meta model for Requirements Engineering in Product Family Context,* Departamento de Ingeniería de Sistemas Telemáticos de la Universidad Politécnica de Madrid, Informe técnico.

[Chan98] Chan, W., Anderson, R., Beame, P., Burns, S., Modugno F., Notkin, D. Reese, J. (1998) *Model Checking Large Software Specifications* IEEE Transactions on Software Engineering, Vol 24, No 7, Julio 1998

[Cho78] Chow, T (1978) *Testing Software Design Modeled by Finite State Machines*, IEEE Transactions on Software Engineering, Vol 4, No 3, pp 178-187

[CKP96] Cremer, J., Kearney, J., Papelis, Y., (1996) *Driving Simulation:Challenges for VR Technology*. IEEE Computer Graphics And Applications. Vol. 16, No. 5:Septiembre 1996, pp. 16-20

[ClNo01] Clements, P., Northrop, L., (2001) *Software Product Lines: Practices and Patterns,* Addison Wesley, Massachussets

[Cou00] Coulter, A. (2000) *Graybox Software Testing in the Real World in Real time*, Paper

[Cu98] Cubranic, D, S. Booth, K.,(1998) *Coordinating Open-Source Software Development*, Proceedings of the IEEE 8th International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises

[CUH99] Cook, D.,Urban, J. Hamilton, S.(1999) *Unix and Beyond: An Interview with Ken Thompson*. IEEE Computer, Mayo 1999, pp. 58-64.

[DaHa99] Damm, W., Harel, D. (1999) *LSCs: Breathing Life into Message Sequence Charts* Proceedings 3rd IFIP International for Open Object Based Distributed Systems, Kluwer Academic, New York.

[Dam99] Dammag, H., and Nissanke, N., (1999) *Safecharts for Specifying and Designing Safety Critical Systems.* 18th {IEEE} Symposium on Reliable Distributed Systems, Lausanne. October, 1999.

[Day93] Day, N. (1993) *A Model Checker for Satatecharts (Linking CASE Tools with Formal Methods)*. Technical Report 93-95. Department of Computer Science. University of British Columbia. Canada.

[DM04] Dueñas, J.C., Mellado, J.,Cerón, R., Arciniegas, J. L., Ruiz, J. L., Capilla, R., (2003) *Model driven testing in Product Family Context*. First European Workshop on Model Driven Architecture with Emphasis on Industrial Application, March 17-18, 2004. University of Twente, Enschede, The Netherlands.

[Dol98] Dolan, T. Weterings, R., Wortmann, J. C. (1998). *Stakeholders in Software-System Family Architectures. Development and Evolution of Software architectures for Product families*. Second International ESPRIT ARES Workshop, Las Palmas de Gran Canaria, Spain, 1998, Proceedings. Springer Lecture Notes in Computer Science 1429

[Dou98] Douglass, B. P.(1998) *Real Time UML. Developing Efficient Objects for Embedded Systems*. Massachusetts: Addison Wesley.

[Dou99] Douglass B. P.(1999) *Doing Hard Time. Developing Real-time systems with UML Objects, Frameworks and Patterns*. Massachusetts: Addison Wesley.

[Du01] Dueñas, J. C., El Kaim, W., Gacek, C. (2001) *Style, structures and views for handling commonalities and variabilities.* ESAPS Deliverable (WG 2.2.3*)* Disponible en www.esi.es

[DuA03] Dueñas, J. C., Arciniegas, J. L., Ruiz, J. L., Cerón, R., Martínez, V., (2003) *An architectural Analysis Tool based on RMA for Real-Time Systems* Departamento de Ingeniería de Sistemas Telemáticos de la Universidad Politécnica de Madrid, Paper.

[Due97] Dueñas, J. C., Rendón, A. y de Miguel, M. A. (1997). Integrated Validation of Real-Time System Models. 9th Euromicro Workshop on Real-Time Systems, 1997. IEEE Computer Society Press, 1997.

[DuOl98] Dueñas, J. C., Oliveira, W. L., de la Puente, J. A. (1998) *A Software Architecture Evaluation Model*. Development and Evolution of Software architectures for Product families. Second International ESPRIT ARES Workshop, Las Palmas de Gran Canaria, Spain, 1998, Proceedings. Springer Lecture Notes in Computer Science 1429

[EBA94] Eisenbahn-Bundesamt (1994). Technische Anforderungen an Sicherungsanlagen der Elektronik. Vorgaben zum Einsatz der Programmiersprache C. Referencia Mü 8004, Parte 42 720, Munich

[EBA99] Eisenbahn-Bundesamt (1999). Technische Anforderungen an Sicherungsanlagen der Elektronik. *Vorgaben zum Einsatz der objektorientierten Programmiersprache C++*. Referencia Mü 8004, Parte 42 730, Munich

[EC96] Comisión Europea (1996) *Technical Specification for Interoperability*. Directiva comunitaria 96/48/EC

[Ei02] Eisenbach, T., Koschke, R., Simon, D., *A Formal Method For The Analysis Of Product Maps* REPL 02, International Workshop on Requirements Engineering for Product Lines, September 2002, Essen, Germany. Disonible en www.research.avayalabs.com/techreport.html

[Ek95] Ek, A. (1995) *The SOMT Method*. Telelogic AB, Paper .

[ElF00] El-Far, I., Whittaker, A., (2000) *Model based software Testing*. Paper disponible en www.model-based-testing.org

[EMM02] Elorriaga, A., Maurolagoitia, I., Mellado, J. (2002) *A Test Automation Experience In The Railway Traffic Control Domain*. ICS Test Conference, April 2002, Düsseldorf, Germany

[ESI02] European Software Institute (2002) *Formalism to describe a domain model based on user specifications using VSL,* paper disponible en http://www.modeldrivenarchitecture.esi.es

[Few99] Fewster, M. Graham, D. (1999*) Software Test Automation. Effective use of test execution tools*. Addison-Wesley, Nueva York

[Fi02] Fischer, K., Vogel-Heuser, B.,(2002) *UML in der automatisierungstechnischen Anwendung – Stärken und Swächen* Automatisierungstechnische Praxis 44, Heft 10.

[Flo97] Flora-Holmquist, A., Morton, E., O'Grady, J. D., Staskauskas M. G. (1997) *The Virtual Finite-State Machine Design and Implementation Paradigm*. Bell Labs Technical Journal. Winter 1997.

[Fow99] Fowler, M. (1999) *UML Distilled: Applying the standard Object modelling language*. 2ª edición, Addison-Wesley, Massachusetts

[Fri92] Fringuelli, B., Lamma, E., Mello, P., Santocchia, G., (1992) *Knowledge-Based technology for Controlling Railway Stations*, IEEE Expert, pp. 45-52

[Gep02] Geppert, C., Schmid, K., (2002) *Proceedings REPL 02. International Workshop on Requirements engineering for Product Lines*. Essen, Alemania. Disponible en www.research.avayalabs.com/techreport.html

[GHJV95] Gamma, E., Helm R., Johnson R. y Vlissides J. (1995). *Design Patterns. Elements of Reusable Object-Oriented Software*. Massachussetts:Adison-Wesley

[Go01] Gomaa, H., (2001) *Modelling Product Lines with UML*, Department of Information and Software Engineering, George Mason University, Paper.

[God96] Godefroid P., Peled D. y Staskauskas M. (1996) *Using partial-order methods in the formal validation of industrial concurrent programs*. IEEE transactions on Software Engineering, vol. 22, no. 7, Julio 1996, pp. 496-507

[Gre00] McGregor, John D. (2000) "*Building Reusable Test Assets for a Product Line*"

tutorial, First Software Product Line Conference. Pittsburgh, Pa.: Software Engineering Institute, Carnegie Mellon University, 2000.

[GrSy01] McGregor, J. D., Sykes, D., (2001) *A practical Guide to Testing Object-Oriented Software,* Addison-Wesley, Massachussets

[Hae99] Haeusler, E., da Fontoura, M. F. (1999) *Using Transition Systems to Formalize a Pattern for Time Dependable Applications* Fourth International Workshop on Object-Oriented Real-Time Dependable Systems 27 – 29 Enero, Santa Barbara, California

[HaGe96] Harel, D., Gery E. (1996) *Executable Object Modelling with Statecharts*. IEEE Proceedings.18 International Conference on Software Engineering (ICSE96), Berlin 1996, pp 246-257

[HaGe97] Harel, D., Gery, E. (1997) *Executable Object Modelling with Statecharts*. IEEE Computer. Julio 1997, Volumen 30, No. 7, pp. 31-42

[HaNa96] Harel, D., Naamad, A. (1996) *The STATEMATE semantics of statecharts*. ACM Transactions on Software Engineering and Methodology 5, Octubre 1996, pp 293-333

[HaPo98] Harel, D. y Politi, M. (1998). *Modelling Reactive systems with Statecharts*. Nueva York:McGraw –Hill.

[Har01] Harel, D. (2001) *From Play-In Scenarios to Code: an Achievable Dream*. IEEE Computer, Enero 2001, pp. 53-60

[Har87] Harel, D. (1987) *Statecharts.A Visual Formalism for Complex Systems*. Science of computer Programming 8 pp. 231-274

[Har88] Harel, D. (1988) *On Visual Formalisms*. Communications of ACM 1988. 31:5 pp. 514-530.

[Har90] Harel, D., Lachover, H., Naamad, A., Pnueli, A., Politi, M., Sherman, R., Shtull-Trauring, A. y Trakhtenbrot, M. (1990) *STATEMATE:A working Environment for the Development of Complex Reactive Systems*. IEEE Transactions on Software Engineering, Vol 16, No. 4 403-413.

[Har99] Harel, D., Kugler, H., (1999) *Synthesizing State-Based Object Systems from LSC Specifications*, The Weizmann Institute of Science, Rehovot, Israel Preliminary version

[Hat95] Haton L. (1995). *Safer C. Developing Software for High-integrity and Safety-critical Systems*. Berkshire,Inglaterra:McGraw-Hill.

[He03] Heckel, R., Lohman, M., (2003) *Towards Model Driven Testing*, Electronic Notes in Theoretical Computer Science 82 no 6 2003.

[Hei96] Heimdahl, M. P.E., Leveson, N. G.,(1996) *Completeness and Consistency in Hierarchical State-Based Requirements*. IEEE Transactions On Software Engineering, Vol. 22, No. 6: Junio 1996, pp. 363-377

[Het84] Hetzel, W. C.,(1984) *The Complete Guide to Software testing*. QED Information Sciences. Wellesley, Massachussets,

[Hol99] Holzmann, G., Smith, M. (1999) *A practical method for verifying event-driven software*. IEEE Proceedings.21 International Conference on Software Engineering, Los Angeles 1999, pp 246-257

[Hue95] Huecas, G. (1995) *Contribución a la formalización de la fase de ejecución de las pruebas* ETSIT UPM; Tesis Doctoral.

[ISO91] ISO (1991) *ISO 9126. Software product evaluation: Quality characteristics and guidelines for their use*. Geneva, Siwtzerland

[ISO89] ISO (1989) *ISO 8807 LOTOS a formal description technique based on the temporal ordering of observational behaviour*. Geneva, Switzerland.

[ITU95] International Telecommunication Union –Telecommunication Standardization Sector (1995) *Recommendation X.290. OSI conformance Testing Methodology and framework for Protocol recommendations for ITU-T Applications General Concepts.*

[ITU96] International Telecommunication Union –Telecommunication Standardization Sector (1996) *Recommendation Z.120.Message Sequence Charts*

[ITU97] International Telecommunication Union –Telecommunication Standardization Sector (1998) *Recommendation Z.500 Framework on formal methods in conformance testing*

[ITU92] International Telecommunication Union –Telecommunication Standardization Sector (1998) *Recommendation X.292 TTCN Tree and Tabular Combined Notation*

[Jac94] Jacobson, I. (1994) *Object-Oriented Software Engineering : A Use Case Driven Approach*, Prentice-Hall, New Jersey

[Jec04] Jeckle, M., Rupp, C., Hahn, J., Zengler, B., Queins, S., (2004) *UML 2 glasklar*, Hanser, Munich, Alemania.

[Jep00] Jepsen, H. P., Nielsen, F. (2000) *A two Part Architectural Model as basis for Frequency Converter Product families.* Third International Workshop on Software Architectures for Product Families IWSAPF-3, Las Palmas, Spain. Springer Lecture Notes in Computer Science 1951

[Ka02] Kang, K., Lee, J., Donohoe, P., (2002) *Feature-Oriented Product Line Engineering* IEEE Software July/August 2002 pp. 50-57

[Ka90] Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A. (1990) *Feature-Oriented Domain Analysis (FODA) Feasibility Study* Technical Report CMU/SEI-90-TR-21 ESD-90-TR-222

[Ki00] Kim, J-M, Porter, A., Rothermel, G., (2000) An empirical Study of Regression Test Application Frequency *,* Proceedings of the International Conference on Software Engineering ICSE 2000, pp. 126-135

[Kna00] Knauber, P., Muthig, D., Schmid, K., Widen, T. (2000) *Applying Product Line Concepts in Small and Medium-Sized Companie,* IEEE Software, September/October 2000, pp 88-95

[Kna01] Knauber, P., Bermejo, J., Böckle, G., Sampaio, J. C., van der Linden, F., Northrop, L., Stark, M., Weiss, D., (2001) *Quantifying Product Line Benefits*, Proceedings of the 4th International Workshop on Product Family Engineering, Springer Lecture Notes in Computer Science 2290

[Koe01] König, N., Bayley, C., (2001) *Eurointerlocking promises lower life-cycle costs* Railway Gazette International Octubre 2001

[Kru01] Krueger, C. (2001) *Easing the Transition to Software Mass Customization*, Proceedings of the 4th International Workshop on Product Family Engineering, Springer Lecture Notes in Computer Science 2290

[KTW98] Knor, R., Trusmuth, G., Weidl, J., *Reengineering C/C++ Source Code by Transforming State Machines*. 2nd International ESPRIT ARES workshop on Developement and Evolution of Software Architectures for product families Las Palmas, Spain, Febero 1998, Springer Lecture Notes in Computer Science 1429.

[Kuu00] Kuusela, J. Savolainen, J. (2000) *Requirements Engineering for product families*. Proceedings of the International Conference on Software Engineering ICSE 2000 pp 61-69., Dublín, Irlanda.

[Lai00] Laibarra, B., Vega, F. (2000) *A systematic approach for the validation of an electronic interlocking*. ICS Test Conference, April 2000, Bonn, Germany

[Ler01] Lerchundi, R. (2001) *System Family process frameworks.* ESAPS Deliverable (WG 2.1.2*)* Disponible en www.esi.es

[Lev95] Leveson, N., (1995) *Safeware. System Safety and Computers*. Addison-Wesley, Massachussets.

[Lin00] van der Linden, F., Obbink, H., (2000) *ESAPS – Engineering software Architectures, Processes, and Platforms for System Families* Third International Workshop on Software Architectures for Product Families IWSAPF-3, Las Palmas, Spain. pp 244-252 Springer Lecture Notes in Computer Science 1951

[Lio02] Lions, J. L., (2002) *Ariane 5 Flight 501 Failure Report by the Inquiry Board*, http://java.sun.com/people/jag/Ariane5.html

[Lut01] Luttikhuizen, P., (editor) (2001) *Requirements Modelling and Traceability*. Public Results of ESAPS Project, Paper recogido en www.esi.es

[Ma95] Maters, W. M. (1995) *Real Time Computing Cornerstones, a System Engineer´s View*. Proceedings 3rd Workshop on Parallel and Distributed Real Time Systems WPRDTS95

[MaBr99] Malan, R., Bredemeyer, D. (1999) *Functional Requirements and Use Cases*. Bredemeyer Consulting, Paper recogido en www.bredemeyer.com

[Mac96] Macala. R., Stuckey, L., Gross, D. (1996) *"Managing Domain-Specific Product-Line Development"*. IEEE Software Mayo 1996, pp.57-67

[MAP97] de Miguel M. A., Alonso, A. y de la Puente J. A. (1997). *Object-oriented Design of Real-time systems with Stereotypes*. 9 Euromicro Workshop on Real-Time Systems. IEEE Computer Society Press

[MeDu00] Mellado, J., Sierra, M., Romera, A. I., Dueñas J. C. (2000) *Railway-Control Product Families: the ALCATEL TAS Platform experience*. Third International Workshop on Software Architectures for Product Families IWSAPF-3, Las Palmas, Spain. Springer Lecture Notes in Computer Science 1951.

[MeDu01] Mellado, J., Dueñas, J. C. (2001) *Automated Validation Environmentfor a Product Line of Railway Traffic Control Systems* Proceedings of the 4th International Workshop on Product Family Engineering, Springer Lecture Notes in Computer Science 2290.

[MeET02] Mellado, J., Elorriaga, A., de la Torre Binimelis, N. (2002) *Automatisierte Validationsumgebung für ein sicheres Bahnsteuerungssystem* Signal+Draht 1+2 pp. 37-38

[Mel04] Mellor, S., (2004) *Agile MDA,* paper disponible en http://www.omg.org/

[Men99] Mendes da Silva Filho, A., Toshiraru, J., de Souza, I., (1999) *Real Time Supervisor Modelling for Telecom Systems.* IEEE Symposium on Application-Specific Systems and Software Engineering & Technology 24 – 27 Marzo, 1999 Richardson, Texas

[MLS97] Mikk, E., Lakhnech, Y., and Siegel, M. *Hierarchical automata as model for statecharts*. Asian Computing Science Conference (ASIAN'97), volume 1345 of LNCS. Springer Verlag, Diciembre 97.

[MLSH98] E. Mikk, Y. Lakhnech, M. Siegel, and G. J. Holzmann. *Implementing Statecharts in Promela/SPIN*. Workshop on Industrial Strength Formal Specification Techniques (WIFT98)

[Mor02] Morris, J., Lee, G., Parker, K., Bundell, G. A., Peng, C. (2002) *Software Component Certification* IEEE Computer pp. 30-36

[Mye79] Myers, J. G. *The art of software Testing*, Nueva York, Wiley,1979

[Neb02] Nebut, C., Pickin, S., Le Traon, Y., Jezequel, J. M., (2002) *Reusable Test requirements for UML Modeled Product Lines.* REPL 02, International Workshop on Requirements Engineering for Product Lines, September 2002, Essen, Germany. Disponible en www.research.avayalabs.com/techreport.html

[Nor00] Nord, R. (2000) *Meeting the Product Line Goals for an Embedded real Time System* Third International Workshop on Software Architectures for Product Families IWSAPF-3, Las Palmas, Spain. pp.19-29, Springer Lecture Notes in Computer Science 1951

[Nus01] Nuseibeh, B. (2001) *Weaving together Requirements and Architectures*. IEEE Computer Marzo 2001. pp. 115-117

[OM03] OMG (2003) *UML Testing Profile*, paper disponible en http://www.omg.org/

[OMG03] OMG (Object Management Group) (2003). *OMG UML v2.0*. Accesible en www.rational.com y en www.omg.org

[Omm00] van Ommering, R., van der Linden, F., Kramer, J., Magee, J. (2000) *The KoalaComponent Model for Consumer Electronics Software* IEEE Computer, Marzo 2000 pp. 78-85

[Pa02] Pachl, J. (2002) *Systemtechnik des Schienenverkehrs* Taubner Verlag, Stuttgart.

[Pac02] Pachl, J. (2002) *Railway Operation and Control* VTD Rail Publishing, Mount Lake Terrace, USA

[Per98] Perry, D. (1998) *Generic Architecture Descriptions for Product Lines*. Second International ESPRIT ARES Workshop, Las Palmas de Gran Canaria, Spain, 1998, Proceedings. Springer Lecture Notes in Computer Science 1429

[Perr95] Perry, W. (1995) *Effective Methods for Software testing"*.Nueva York, John Wiley & Sons.

[PMF99] Paulo, F., Masiero, P., Ferreira de Oliveira, M. (1999) *Hypercharts: Extended Statecharts to support Hypermedia Specification* IEEE transactions on Software Engineering, Vol 25, No 1, Enero/Febrero 1999

[Pos01] Postema, H., Obbink, H., (2001) *Platform Based Product Development*, Proceedings of the 4th International Workshop on Product Family Engineering, Springer Lecture Notes in Computer Science 2290

[Pos98] Poston, R. (2001) *Automating Specification-Based Software Testing*, IEEE Computer Society Press.

[Qua98] T. Quatrani, *Visual Modeling With Rational Rose and UML*. Addison-Wesley Object Technology Series. Addison Wesley, 1998.

[Rat03] Rational Corporation (2003) *UML Version 2.0*. Accesible en www.rational.com y en www.omg.org

[Kru03] Krutchen, P. (2003) The *Rational Unified Process: An Introduction*. Addison-Wesley Massachussets.

[Que89] Quemada, J., Pavón, S., Fernández, A. (1989) *Transforming LOTOS specifications with LOLA: The parameterized expansion*. In Kenneth J. Turner, editor, Proc. Formal Description Techniques I, pages 45-54. North-Holland, Amsterdam, Netherlands.

[Ray98] Raymond, E., (2001) The Cathedral and the Bazaar, O'Reilly, California

[Red01] Reddy, R., (2001) *Testing E-Business Applications without Breaking the Bank*. Cutter IT Journal, Septiembre 2001, Vol 14, No 9, pp 20-28

[ReMo03] Rentsch, U., Moik, A. (2003) *Validation of EBICAB 2000 acording to CENELEC* Signal+Draht 95 1+2/2003, pp. 28-33

[Ren97] Rendón, A. (1997) *Prueba Incremental de Modelos de Sistemas de Tiempo Real*. Universidad Politécnica de Madrid, Escuela Técnica Superior de Ingenieros de telecomunicación, Tesis Doctoral.

[Ru98] Rubaai, A., Kotaru, R., Branch, R. H., Hussein, A., (1998) *Design of a neuroclassifier / detector for Amtrak rail-road track operations*. Dept. of Electr. Eng., Howard Univ., Washington, DC; Industry Applications Conference, 1998. Thirty-Third IAS Annual Meeting.

[RuH01] Rupp, C., Hruschka, P. (2001) *Echt-Zeit für Use-Cases*. Paper recogido en www.sophist.de

[Rum91] Rumbaugh, J., et al. (1991) *Object-Oriented Modelling and Design*, Addison-Wesley, Massachusetts

[SaCa95] Sane, A. Campbell, R., (1995) *Object Oriented Machines Subclassing, Composition, Delegation and Genericity*. OOPSLA 95 pp. 17-32

[Sat02] Sathayaja, N. J. (2002) *V&V Tests for Embedded Flight Control Software*. 3rd ICSTEST International conference on Software Testing, Abril 2002, Düsseldorf, Alemania

[Sch00] Schmidt, M. Fernández, J. (2000*) Integriertes Testmanagement sichert Qualität in Software Projekten*, Signal+Draht, Junio 2000.pp 17-18

[Sch01] Schmidt, K., (2001) *An Initial Model of Product Line Economics,* Proceedings of the 4th International Workshop on Product Family Engineering, Springer Lecture Notes in Computer Science 2290

[Sch02] Schmid, K., Verlage, M., (2002) *The economic impact of Product Line Adoption and Evolution* IEEE Software July/August 2002 pp. 50-57

[Scho97] Scholz, P., Nazareth, D. (1997) *Communication Concepts for Statecharts: A semantic Foundation*. 4th International AMAST Workshop on Real-Time Systems ans Concurrent and Distributed Software. ARTS'97 Palma, Mallorca, España, Mayo 1997

[SDL95] International Telecommunication Union(1995), CCITT *Specification and Description Language (SDL)*, ITU-T Recommendation Z.100

[SEI01] Software Engineering Institute (2001) *A Framework for Software Product Line Practice – Version 3.0.* Software Engineering Institute, Carnegie Mellon University. http://www.sei.cmu.edu/plp/framework.html

[SEI03] CMMI, disponible en www.sei.org

[SEI91] Paulk, M., Curtis, B., Chrissis, M. B., Weber, C., (1991) *Capability Maturity Model for Software, Version 1.1,* disponible en www.sei.org

[Sel93] Selic, B., (1993) *An Efficient Object-Oriented Variation of the Statecharts Formalism for Distributed Real-Time Systems*. CHDL '93: IFIP Conference on Hardware Description Languages and Their Applications, Ottawa, Canada

[Sel98] Selic B., Rumbaugh J., (1998). *Using UML for Modeling Complex Real-Time Systems*, Object Time Ltd. (paper) Marzo, 1998

[Sel99] Selic, B. (1999) *Protocols and Ports: Reusable Inter-Object Behavior Patterns* Second IEEE International Symposium on Object-Oriented Real-Time Distributed Computing 2 – 5 de Mayo, Saint-Malo, France

[Sik01] Sikula, C., Sieverding, P. (2001) *SIMIS LC Teststrategie für eine Anwendungssoftware auf Basis des CENELEC Entwicklungsprozesses*, Signal+Draht (93) 7+8 2001, pp 10-16

[Sow98] Sownya A., Ramesh S., (1998) *Extending Statecharts with temporal logic*. IEEE Transactions On Software Engineering Vol. 24, No. 3: JUNE 1998, pp. 216-230

[Ste99] Stevens, P. (1999) *UML for describing Product Line Architectures?* División de Informática, Universidad de Edimburgo, paper.

[Tel02] Telelogic (2002) *TTCN-3 The Ultimate Test. The evolution of a multi-purpose, standarized test language*, Paper disponible en www.telelogic.com

[Th01] Thane, H., Petterson, A., Hansson, H., (2001) *Integration of Fixed Priority Scheduled Real-Time Systems*. IEEE Real Time Embedded system Workshop.

[Tol01] Tolvanen, J. P. (2001) *Domänenspezifische Modellierungssprachen Für Produktfamilien (Domain-Specific Modeling Languages for Product Families)*, ObjectSpektrum, August/September, 2001.

[UN00] UNISIG Working Group (2000) *ERTMS/ETCS Functional Requirements Specifications.* Disponible en www.unife.org

[Val00] del Valle Alvarez, J (2000). *Elektronisches Stellwerk ESTW L90 5 in Spanien. Electronic interlocking ESTW L90 5*. Signal+Draht 04/2000.

[vdB94] von der Beeck, M. (1994), *A Comparison Of Statecharts Variants*. Volume 863 of Lecture Notes in Computer Science, pp 128-148, Alemania, Springer

[Vin99] Vinga-Martins, R. (1999) *Requirements Traceability for Product Lines*, Paper recogido en www.esi.es

[Vli98] Vlissides, J. (1998) *PATTERN HATCHING. Design Patterns Applied* Massachusetts, Addison-Wesley

[We03] Weber, M., Weisbrod. J. (2003) *Requirements Engineering in Automotive Development: Experiences and Challenges*, IEEE Software, January February 2003, pp 2-10.

[Wei01] Weingartner, J. (2001) *Product Family Engineering and testing in the medical domain – validation aspects*, Proceedings of the 4th International Workshop on Product Family Engineering, Springer Lecture Notes in Computer Science 2290

[Wei97] Weidl J., Klösch R, Trausmuth G, y Gall H. (1997) *Facilitating program comprehension via generic components for state machines*. 5th Workshop on Program Comprehension (Dearborn, Michigan, USA), pp 118-27. IEEE Computer Society Press, Mayo 1997

[Wh00] Whittle, J., Schumann, J. (2000) *Generating Statecharts Designs from Scenarios* Proceedings of the International Conference on Software Engineering ICSE 2000 pp 61-69., Dublín, Irlanda

[Zav96] P. Zave, M. Jackson (1996) *Where do operations come from? A multiparadigm specification technique*.. IEEE Transactions on Software Engineering. Vol. 22, No. 7 Julio 1996, pp. 508-528

Curriculum Vitae

2002-2004	Jefe de Departamento de Sistemas ATP (Automatic Train Protection), desplazado en Berlín (Alemania) como encargado de la transferencia de tecnología del sistema ETCS2000 en colaboración con Alcatel Alemania.
2000-2002	Ingeniero de validación y verificación de software en el departamento de enclavamientos electrónicos de Alcatel TAS.
1998-2000	Ingeniero de desarrollo responsable de la migración de plataforma hardware y del sistema operativo del sistema INTERSIG 905 colaborando con Alcatel Austria.
1995-1998	Participación como ingeniero de desarrollo en la especificación, diseño e implementación del enclavamiento electrónico INTERSIG 905.
1994-1995	Ingeniero de desarrollo de software para el sistema de control ferroviario ENCE L90 de la parte de control de elementos de hardware en distintos proyectos en España, Portugal, Polonia e Israel.
1993-1994	Encargado como Ingeniero de desarrollo de la transferencia de tecnología en Alcatel TAS de una parte del enclavamiento electrónico ENCE L90 en colaboración con Alcatel Alemania.
1986-1992	Ingeniero Superior de Telecomunicación por la Universidad Politécnica de Madrid. Calificación: NOTABLE. Proyecto de Fin de Carrera realizado en colaboración con la empresa TELDAT S.A. Calificación: MATRICULA DE HONOR.